Facultad de Física
Grado en Física

Trabajo Fin de Grado

Implementación de una Red Neuronal mediante el Lenguaje Python

---

Autor:
José Damián García Noya

Tutor:
Pablo García Tahoces
Área de Ciencias de la Computación e Inteligencia Artficial
Ídepartamento de Eléctrónica y Computación

Julio 2020

Resumen

Las redes neuronales artificiales son uno de los modelos de aprendizaje automático más innovadores de las últimas décadas y permiten resolver cantidad de problemas en diferentes ámbitos, entre ellos la física. En este trabajo se pretende dar una explicación completa del funcionamiento de las redes neuronales tradicionales poniendo énfasis en la arquitectura y el proceso de aprendizaje en las mismas. Adicionalmente se aporta una aplicación desarrollada por medio del lenguaje Python que permite al usuario definir y utilizar su propia red neuronal así como realizar una serie de tareas extra relacionadas con el preprocesado de los datos. Finalmente utilizando conjuntos de datos típicos asociados a tareas de regresión (Viviendas Boston) y a tareas de clasificación (Flor Iris) se lleva a cabo una comparativa del rendimiento de una red neuronal construida por medio de la aplicación anterior y una red neuronal generada a través de una popular librería de aprendizaje automático en lo que a redes neuronales se refiere (Keras).

Resumo

As redes neurais artificiais son un dos modelos de aprendizaxe automática máis innovadores das últimas décadas e permiten resolver cantidade de problemas en diferentes ámbitos, entre eles a física. Neste traballo preténdese dar unha explicación completa do funcionamento das redes neurais tradicionais poñendo énfase na arquitectura e no proceso de aprendizaxe das mesmas. Adicionalmente apórtase unha aplicación desenvolvida por medio da linguaxe Python que permite ao usuario definir e utilizar a súa propia rede neural así como realizar unha serie de tarefas extra relacionadas co preprocesado dos datos. Finalmente utilizando conxuntos de datos típicos asociados a tarefas de regresión (Vivendas Boston) e a tarefas de clasificación (Flor Iris) lévase a cabo unha comparativa do rendemento dunha rede neural construída por medio da aplicación anterior e unha rede neural xerada a través dunha popular librería de aprendizaxe automático no que a redes neurais se refire (Keras).

Abstract

Artificial neural networks have been one of the most innovative models in machine learning in the last decades and they allow to solve many problems in different areas, one of which is physics. In this dissertation the intention is to provide a full explanation about the behavior of neural networks placing the emphasis in the net's architecture and the net's learning process. Moreover an app developed through the Python language is given, which allows the user to define and use his own neural network as well as do a series of tasks related to the data preprocessing. Finally using common datasets associated with regression tasks (Boston Housing) and classification tasks (Iris Flower) the performances of a neural network constructed with the previous app and a neural network developed with a popular machine learning and neural network library (Keras) are compared.

Agradecimientos

Me gustaría agradecerle a mi tutor Pablo su ayuda, sin la cual este trabajo no habría sido posible. También quisiera agradecerle de corazón a mis padres todo su apoyo durante estos cuatro años, mis éxitos son también suyos.

Índice

1. Introducción

En la actualidad las denominadas técnicas de "machine learning" o aprendizaje automático son cada vez más relevantes tanto en el ámbito de la investigación como en la práctica en el sector privado. Esto se debe, en gran parte, a la disponibilidad creciente de cantidades masivas de información de diversa índole que permiten hacer predicciones y reconocer patrones en diferentes procesos. El rango de aplicabilidad de este tipo de técnicas es amplísimo y se utilizan activamente en áreas relacionadas con las matemáticas, la ingeniería, las finanzas, el sector de la salud y también la física entre otros.

El aprendizaje automático es como tal una disciplina de las ciencias de la computación y representa una de las ramas más extendidas de la inteligencia artificial. En los últimos años se ha propagado una desafortunada componente de marketing sobre estos métodos que introduce cierto esoterismo sobre su funcionamiento pero es necesario incidir en el hecho de que estas técnicas son esencialmente algoritmos matemáticos originados bajo el prisma de la estadística y que en consecuencia pueden estudiarse sistemáticamente con total rigor formal.

Como ya se ha comentado con anterioridad el aprendizaje automático permite realizar predicciones y reconocer patrones en diversos procesos. Para conseguirlo estas técnicas requieren de la utilización de datos o variables que pueden ser tanto numéricas como categóricas. Por medio de estos datos los algoritmos ajustan sus parámetros, proceso que se conoce como aprendizaje. Naturalmente cada algoritmo tiene una forma particular de realizar el ajuste de estos parámetros internos. Una vez hecho esto el modelo está listo para utilizarse en las tareas pertinentes.

En cuanto a la naturaleza de los datos que se utilizan para el aprendizaje se puede distinguir entre aprendizaje supervisado y aprendizaje no supervisado. En los métodos de aprendizaje supervisado el algoritmo es una función que, a partir de unos datos de entrada, produce un resultado o salida. Los datos de entrenamiento utilizados para el aprendizaje en este caso tienen la peculiaridad de que se conoce a priori el resultado o salida deseada. Para los métodos de aprendizaje no supervisado sin embargo no se conoce el resultado esperado para los datos que se utilizan en el proceso de entrenamiento y el aprendizaje se lleva a cabo por medio del agrupamiento y/o reconocimiento de patrones en los datos.

Por otra parte en lo que se refiere al tipo de variable que representa la salida de un algoritmo de aprendizaje supervisado se pueden diferenciar tareas de regresión y de clasificación.

Los algoritmos de clasificación se utilizan cuando el resultado que se busca es una variable categórica, es decir, la respuesta al problema tiene un conjunto finito de resultados o etiquetas posibles. Si los datos sobre los que se trabaja pueden englobarse en únicamente dos clases se dice que se trata de un problema de clasificación binaria mientras que si hay más de dos entonces se habla de clasificación multicategórica. Un ejemplo clásico de clasificación podría ser determinar, a partir del tamaño de los pétalos de una flor, a qué tipo de especie pertenece la misma. Nótese que matemáticamente puede especificarse la

pertenencia o no a una clase por medio de una variable binaria, dicho de otro modo, con ceros y unos.

Los algoritmos de regresión por su parte se utilizan cuando el resultado buscado es una variable numérica. Un ejemplo clásico de problema de regresión sería el de predecir el precio de una vivienda a partir de ciertos parámetros acerca de la misma como la antigüedad, la localización, etc. El algoritmo de regresión más simple y popular quizás sea la regresión lineal que permite obtener una relación lineal entre los datos de entrada y la salida.

Las redes neuronales artificiales (ANN) son uno de los muchos modelos de aprendizaje automático. Su funcionamiento se basó inicialmente en su homólogo biológico pero con el paso del tiempo y los nuevos avances, las redes neuronales artificiales evolucionaron y en la actualidad se han convertido en uno de los pilares más importantes del aprendizaje automático y la inteligencia artificial. En el año 1943 Warren McCulloch y Walter Pitts descubrieron por primera vez que el funcionamiento de una neurona biológica podía ser reproducido a través de una suma ponderada de señales limitada por un umbral. De esta forma usando la estructura anterior se podían desempeñar funciones lógicas simples. A partir de entonces se hicieron algunas contribuciones relevantes al campo pero el descubrimiento más destacable quizás sea la propuesta en la década de los 90 del algoritmo de propagación hacia atrás del que se hablará más adelante y constituye una de las bases más importantes para el funcionamiento de las redes neuronales tal y como se entienden hoy en día.

Durante el desarrollo de las redes neuronales se descubrieron nuevas arquitecturas que resultaban especialmente útiles en tareas relacionadas con la visión artificial, es decir, la capacidad de una máquina para reconocer o clasificar imágenes. Este tipo de redes conocidas como redes neuronales convolucionales (CNN) comienza a desarrollarse en 1980 cuando Kunihiko Fukushima introduce el 'Neocognitron' basándose en el trabajo de expertos en la estructura del cortex visual del cerebro. La contribución fundamental consiste en añadir a las redes dos nuevos tipos de capas, las capas de convolución y las capas de reducción del muestreo. En esencia estas nuevas capas funcionan como una especie de filtros de selección de características que actúan sobre la imagen de entrada y la simplifican de modo que en la salida se obtienen unas variables que facilitan enormemente la clasificación. El significado de estas variables resultado del filtrado es difícil de determinar y lógicamente esto puede ocasionar problemas en ciertas ocasiones porque el humano no tiene la capacidad de conocer en base a qué características de la imagen se realiza la clasificación.

En 1998 Yann LeCun et al. aplicaron el algoritmo de propagación hacia atrás para entrenar el sistema y demostró su correcto funcionamiento con un clasificador de imágenes (32x32 píxeles) de dígitos escritos a mano. Previamente en 1988 W.Zhang propuso algunas modificaciones en el algoritmo que se aplicaron en el campo de la imagen médica, más concretamente, en la detección automática del cáncer de mama en las mamografías.

A partir de 2004 se gesta la nueva revolución en el campo de las CNN gracias a su implementación en unidades de procesamiento gráfico (GPU). La idea fundamental es que los algoritmos de las CNN pueden paralelizarse através de una GPU y de este modo

se alcanzan velocidades de procesamiento mucho mayores que con una CPU típica. Esto logicamente posibilita el aumento del rendimiento de las redes neuronales. En este sentido en 2012 Alex Krizhevsky y sus colaboradores diseñaron una CNN implementada sobre una GPU que ganó el ImageNet Large Scale Visual Recognition Challenge que consistía en un desafío para clasificar las imágenes de una base de datos llamada IMAGENET demostrando su gran potencial para la visión artificial.

Las redes neuronales se utilizan en muchos ámbitos de la ciencia y la tecnología y la física no es una excepción. En concreto resultan de gran interés siempre que se tenga la necesidad de organizar y extraer información relevante de un espacio multiparamétrico de alta dimensionalidad.

En la física de altas energías las redes neuronales comenzaron a utilizarse tímidamente en 1988. Su uso era bastante esporádico y la comunidad de físicos era algo escéptica con su utilización pero con el paso del tiempo, el incremento de la complejidad y la cantidad de datos generados, las redes neuronales empezaron a utilizarse más a menudo. La primera aplicación de las ANN en este ámbito fue la reconstrucción de trayectorias de partículas en los experimentos (redes neuronales recurrentes) pero en la actualidad la mayoría de aplicaciones están relacionadas con el análisis general de los datos procedentes de los experimentos. Una situación habitual en física de altas energías es que se tengan varios canales de reacción abiertos en un experimento y que sólo se quiera estudiar uno de ellos en concreto. En casos así es esencial poder distinguir los datos asociados al canal de desintegración que se quiere estudiar, que se denomina señal, y el resto de datos, que se denominan generalmente ruido. Para llevar a cabo esto se puede utilizar una red neuronal para la clasificación utilizando datos de entrenamiento generados mediante simulaciones de Montecarlo. El primer proceso de este tipo estudiado con éxito mediante redes neuronales fue el decaimiento del bosón Z. Uno de los principales inconvenientes del uso de redes neuronales en física de altas energías es que cuando el proceso físico no se entiende lo suficientemente bien su uso puede resultar inviable por la falta de datos con los que entrenar el modelo.

En astrofísica los avances tecnológicos recientes han permitido producir grandes cantidades de información y de alta calidad que pueden ser utilizadas en modelos de redes neuronales para multitud de aplicaciones. Las aplicaciones principales en astrofísica se centran en tareas de clasificación de distintos tipos. Tradicionalmente los primeros intentos de utilización de redes neuronales en astrofísica estaban destinados a la separación de estrellas y galaxias o a la clasificación de las galaxias basándose en sus características morfológicas o espectrales. Otro tipo de aplicación quizás un poco más actual es la identificación de objetos astrofísicos de interés como radiogalaxias, cuásares, etc por medio del análisis de datos de las observaciones astronómicas. Por último comentar que también pueden utilizarse con éxito en la física de astropartículas.

En vista del gran potencial que guardan las redes neuronales artificiales en distintos ámbitos de la ciencia y la tecnología se discutirá a continuación el formalismo matemático de una red neuronal multicapa así como los algoritmos necesarios para su implementación en un lenguaje de programación.

2. Arquitectura de una ANN

En esta sección se discuten las principales características de una ANN poniendo enfásis en el formalismo matemático que se utiliza para describirlas correctamente.

2.1. Nociones Previas

Para comprender a nivel fundamental como funciona una red neuronal es necesario en primer lugar tener una visión general de su estructura. Esencialmente una red neuronal puede entenderse como una función que transforma unas variables de entrada en unas variables de salida.

Las variables de entrada pueden tener una naturaleza muy diversa en función del problema que se trata de resolver. En un problema de regresión por ejemplo cada una de las variables representa una característica de la entrada y por lo tanto la manera más adecuada de representarlas matemáticamente es por medio de un vector. Por otro lado para problemas como la clasificación de imágenes, cada variable puede representar el grado de blanco o negro de un píxel de la imagen. En principio estas últimas variables también podrían representarse por medio de un vector pero por motivos relacionados con la generalización del aprendizaje resulta más beneficioso utilizar una representación basada en matrices. En base a esto se ve que la mejor manera de representar las variables de entrada es por medio de un **tensor** de distinto orden según el tipo de problema. Por ejemplo en las CNN generalmente las variables de entrada son tensores de orden 3 donde dos índices se utilizan para determinar el píxel de la imagen y el restante para determinar el color del sistema RGB al que se hace referencia.

Las variables de salida, al igual que las de entrada, también pueden tener una distinta naturaleza en función del problema que se pretende resolver por lo que se representarán generalmente con un tensor. Por ejemplo en problemas de regresión la variable de salida es inequívocamente un escalar mientras que en problemas de clasificación será un vector de diferente dimensión según el número de clases con las que se trabajen.

En base a la consideración anterior de las variables de entrada y salida como tensores puede establecerse inmediatamente que las redes neuronales en sí mismas no son más que una consecución de operaciones tensoriales sobre la entrada que acaban generando una salida. Lo que se conoce como **arquitectura** de la red neuronal es lo que determina en qué consisten estas operaciones tensoriales y como se asocian entre ellas para dar lugar a una salida determinada. Los dos tipos de arquitecturas más importantes son las de las redes neuronales recurrentes (RNN) y las redes neuronales prealimentadas o 'feed-forward'. Las redes neuronales recurrentes se caracterizan porque las operaciones tensoriales siguen la estructura de un grafo secuencial y por lo tanto pueden presentar ciclos en los que se producen bucles de realimentación. Por otro lado las redes neuronales 'feed-forward' como su propio nombre en inglés indica realizan operaciones de manera secuencial sin ningún tipo de bucle de realimentación al contrario que sus homólogas.

A partir de aquí nos centraremos en las redes neuronales 'feed-forward' por ser las más

comunes en la mayoría de aplicaciones prácticas y se utilizarán vectores para representar las variables de entrada y salida por ser los casos de mayor interés sabiendo además que las nociones introducidas son fácilmente generalizables al caso de tensores de orden mayor.

2.2. Concepto de Capa y Neurona Artificial

El concepto más importante para entender la arquitectura de las redes neuronales 'feed-forward' es el de **capa**. Esencialemente las capas de una red neuronal son pequeñas partes o módulos que la componen y que realizan ciertas operaciones tensoriales. La organización de estos módulos es secuencial en forma de capas dando lugar a su nombre de modo que el resultado de las operaciones en una capa se convierte en la entrada para la siguiente capa y así sucesivamente hasta llegar a la salida de la red.

En cuanto a la nomenclatura se considera generalmente que las variables de entrada no conforman una capa y que la última capa de la red es la salida tal y como puede apreciarse en la figura 1. En base a esto las redes neuronales tienen como mínimo una capa aunque generalmente son más. Las capas que no se corresponden con la salida de la red se denominan **capas ocultas** ya que las variables que las ocupan no son generalmente visibles para el usuario ni tampoco tienen un interés específico. Si la red tiene más de una capa oculta se dice que es una red profunda 'deep' y se engloba dentro de lo que se conoce como 'deep learning' aunque en esencia su funcionamiento no se diferencia demasiado de redes con una única capa oculta. Más adelante se comentarán algunos detalles de interés relacionados con el número de capas de la red y las propiedades que le confiere a la misma.

Como ya se ha comentado con anterioridad cada capa de la red neuronal realiza una serie de operaciones tensoriales sobre unas variables y las transforma en otras. Para entender a nivel fundamental estas operaciones conviene compartimentar cada capa de la red en uno o más **nodos** o **neuronas artificiales**.

Cada uno de los nodos de la capa se encarga de calcular el valor de una variable de salida de la capa. Adicionalmente se dice que la red está totalmente conectada porque cada uno de estos nodos utiliza la información de todas y cada una de las entradas para calcular la salida. La figura 2 indica los distintos componentes de una neurona artificial.

- **Entrada y Salida.** Las variables de entrada x_i son un conjunto de n variables numéricas que como su propio nombre indica son el punto de partida del cómputo de la neurona mientras que la variable de salida y es única y representa el resultado de las operaciones realizadas con la entrada.
- **Pesos Sinápticos.** Los pesos sinápticos o simplemente pesos w_i son $n+1$ variables numéricas de las cuales n están asociadas a las variables de entrada y la restante es independiente. La función de los pesos de la neurona es el de determinar la importancia de cada variable de entrada. El efecto concreto depende en cierta medida de la función de activación pero la idea es que generalmente cuanto mayor sea el valor absoluto del peso asociado a una variable, mayor será la influencia que tendrá esta en la salida. De este modo si el peso asociado a una variable es nulo, la salida de la

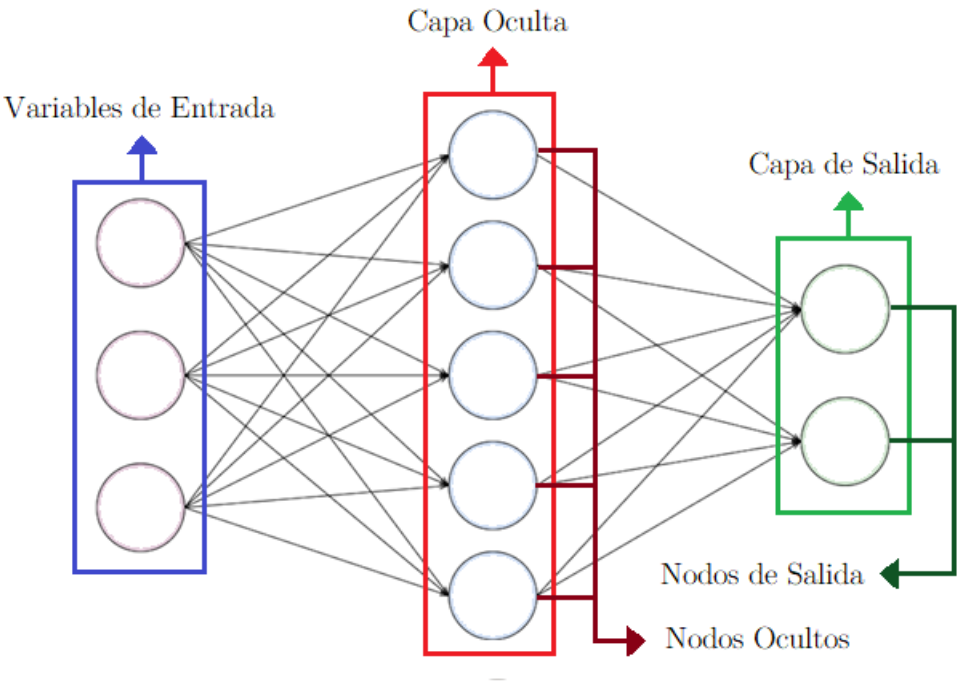


Figura 1: Esquema Red Neuronal de 2 Capas.

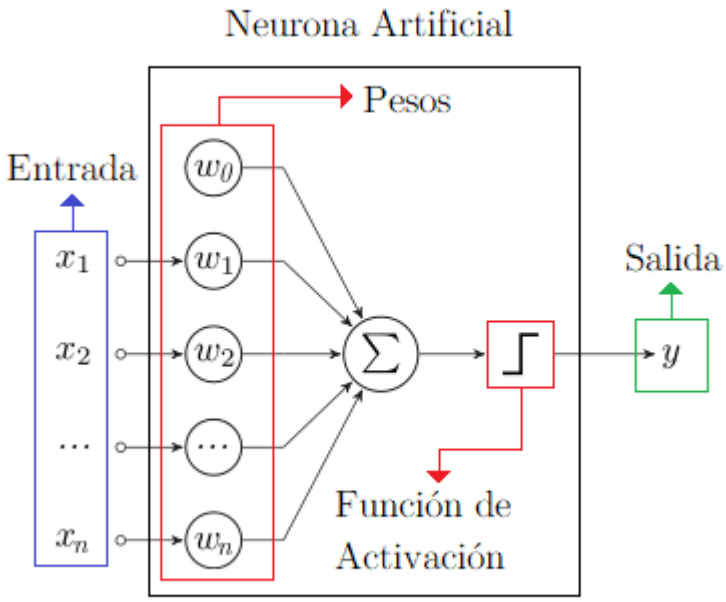


Figura 2: Esquema Neurona Artificial.

neurona no dependerá en absoluto de esta variable y será la misma aunque cambie. Por contra si el peso tiene un valor muy elevado, pequeños cambios en la variable

de entrada causarían variaciones importantes en la salida. El peso independiente permite obtener una salida distinta de cero cuando la entrada es indénticamente nula para todas las variables de entrada.

- **Función de Activación.** La función de activación es uno de los elementos más importantes de la neurona artificial. Se trata de una función matematica f que actúa sobre la variable u resultado de sumar las variables de entrada x_i ponderadas por los pesos w_i. El resultado de la función constituye la salida y de la neurona. Existen muchas formas funcionales que pueden desempeñar el papel de función de activación en una neurona y su elección depende de la tarea que se quiera desempeñar ya que cada una tiene sus ventajas e inconvenientes para cada situación.

Matemáticamente la salida y de una neurona puede expresarse entonces como:

$$y = f(u) = f(w_0 + \sum_{i=1}^{n} w_i x_i)$$

Nótese que la expresión anterior es fácil de implementar utilizando cualquier lenguaje de programación de modo que una computadora puede realizar este tipo de operaciones de manera muy eficiente incluso para un gran número de variables de entrada.

2.3. Funciones de Activación

En lo que sigue se hablará brevemente de algunas funciones de activación de interés tomando en cuenta su forma funcional y sus aplicaciones.

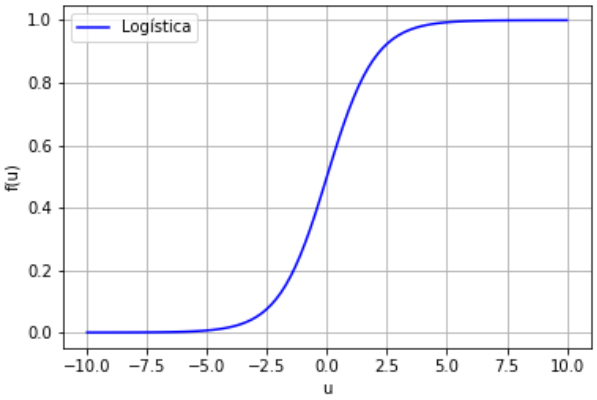
Funcíón Logística

La función logística tiene la forma:

$$f(u) = \frac{1}{1 + e^{-u}}$$

Ampliamente conocida en el ámbito de la estadística, es una de las funciones más utilizadas en las redes neuronales tradicionales por sus propiedades matemáticas.

Figura 3: Función Logística.



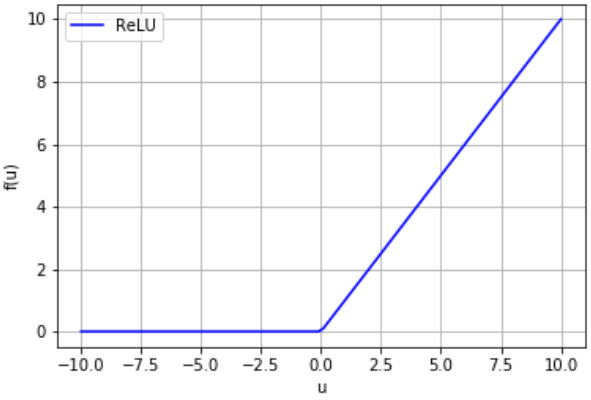
Función Softmax

La función softmax tiene la forma :

$$(y_1, y_2, ..., y_j, ...) = \left(\frac{e^{u_1}}{\sum_{i=1} e^{u_i}}, \frac{e^{u_2}}{\sum_{i=1} e^{u_i}}, ..., \frac{e^{u_j}}{\sum_{i=1} e^{u_i}}, ... \right)$$

Al contrario que su homóloga anterior es una función multivariable dado que para conocer el valor final de cada nodo es necesario saber cual es su valor previo en todos los demás. Esta particularidad permite que el resultado pueda interpretarse como una probabilidad de modo que se usan ampliamente como función de activación en la última capa de la red para problemas de clasificación multicategórica.

Figura 4: Función ReLU.



Función ReLU

La función ReLU (Rectified Linear Unit) tiene la forma:

$$f(u) = max(0, u)$$

Muy popular en redes profundas y utilizada en CNN's por su mejor rendimiento con respecto a la función logística en tareas de visión artificial y reconocimiento de voz.

2.4. Notación General ANN Multicapa

Teniendo en cuenta todo lo expuesto con anterioridad es posible generalizar la notación para referirse al conjunto de la red neuronal. Sea una red neuronal 'feed-forward' con l capas. Considerando la capa k-ésima de la red denotaremos la variable de entrada i-ésima como $x_i^{(k)}$, la variable j-ésima de salida como $y_j^{(k)}$ y la variable j-ésima de salida antes de aplicar la función de activación como $u_j^{(k)}$. Siendo esto así las variables $x_i^{(1)}$ representan el vector de entrada de la red neuronal y las variables $y_j^{(l)}$ representan el vector de salida de la red. Nótese que como las capas están conectadas de manera secuencial se verifica para $k > 1$ que $y_i^{(k-1)} = x_i^{(k)}$.

De esta forma el vector de salida de la capa k-ésima con un vector de entrada de dimensión n puede expresarse como sigue:

$$y_j^{(k)} = f(u_j^{(k)}) = f(w_{0j}^{(k)} + \sum_{i=1}^{n} w_{ij}^{(k)} y_i^{(k-1)})$$

Donde se ha añadido la definición de $y_i^{(0)} = x_i^{(1)}$ para obtener una notación simplificada en la que sólo sea necesario referirse a la variable y. Nótese además que los pesos sinápticos se convierten con esta notación en un tensor de orden 3 ya que cada capa, cada neurona y cada variable de entrada tiene su propio peso distinto de los demás.

2.5. Propiedades Matemáticas

En las secciones anteriores se han descrito en detalle las operaciones tensoriales que transforman un vector de entrada en otro vector de salida en una red neuronal sin embargo todavía no se ha comentado para qué pueden utilizarse. A continuación se discutirán brevemente estos temas y las propiedades matemáticas que lo posibilitan.

En la introducción de este documento se comentó que las redes neuronales funcionan muy bien como herramientas del aprendizaje automático. De manera muy resumida los modelos de aprendizaje automático son herramientas que permiten adaptarse al comportamiento o tendencia de unos datos de entrenamiento y generalizar los resultados a nuevos datos desconocidos para realizar tareas de regresión o clasificación.

Aunque no lo parezca a primera vista las tareas de regresión y clasificación son muy similares en realidad. En un problema de regresión el objetivo es encontrar la forma funcional que se ajusta de la mejor manera a una serie de datos de entrenamiento con sus correspondientes variables. A su vez para un problema de clasificación lo que se busca es separar los datos de entrenamiento para asignarles un determinado atributo o clase según el valor de sus variables. En esencia el procedimiento que se sigue para separar los datos es buscar una forma funcional y un umbral. Si la función aplicada a unos datos queda por debajo del umbral se les asocia una clase mientras que si queda por encima se les asocia otra clase. Lógicamente para tareas de clasificación multiclase esto se hace más complicado pero el concepto es el mismo.

En cualquier caso se ve que tanto para regresión como para clasificación es fundamental encontrar una determinada **forma funcional** que se adapte al problema que se pretende resolver. La capacidad de un modelo para adaptarse a una determinada forma funcional reside generalmente en una serie de parámetros internos del modelo que se adaptan según el problema. Por ejemplo para un modelo de regresión lineal estos parámetros serían la pendiente y el término independiente de la recta. En una red neuronal los parámetros internos variables son los pesos sinápticos de cada una de las neuronas artificiales.

Para que un modelo de aprendizaje automático se considere bueno es muy deseable que pueda utilizarse en un gran número de situaciones diversas. Para garantizar esto último es necesario que el modelo sea capaz de aproximarse a cualquier función continua imaginable con un error tan pequeño como se quiera. Cuando un modelo verifica esta propiedad se dice que es un **aproximador universal**.

Lo que determina que una red neuronal sea un aproximador universal o no es la función de activación utilizada y la arquitectura de la red, es decir, la manera en la que se producen las conexiones entre neuronas. En este sentido se han demostrado diversos teoremas que en ciertas condiciones garantizan que la red neuronal es un aproximador universal. En 1989 George Cybenko fue el primero en demostrar que una red 'feed-forward' con una única capa oculta con suficiente número de nodos es un aproximador universal si y solo si la función de activación no era polinomial. Más adelante se demostró matemáticamente que es suficiente con que se disponga de una capa oculta y que la función de activación sea una función no constante, acotada y continua.

Un inciso muy importante en este aspecto es que lo que garantizan los teoremas que se acaban de comentar es que existen unos valores concretos de los pesos sinápticos para cualquier función continua arbitraria de tal forma que la red se aproxima con el grado de precisión deseado a dicha función. En ningún caso se proporciona ningún método para averiguar cuales son esos valores de los pesos ni tampoco se garantiza que exista alguna forma de encontrarlos. Esto último se discutirá en detalle en el siguiente capítulo.

2.6. Interpretación Geométrica

Como suele suceder con los conceptos matemáticos pueden resultar complicados de entender debido a su notación enrevesada. En la presente sección se ofrece una interpretación geométrica a las operaciones que realiza una red neuronal en una tarea de clasificación lo que permite entender mejor su funcionamiento.

Sean una serie de datos con cierta dimensionalidad y con sus correspondientes variables numéricas. Cada uno de estos datos puede interpretarse como un punto en un espacio de dimensión equivalente al número de variables del problema. El procesado de estos datos por parte de una red neuronal puede entonces interpretarse como una compleja secuencia de transformaciones geométricas sobre estos puntos.

Imagínese un problema de clasificación en el que se disponen de dos clases distintas. La siguiente analogía describe el proceso mediante el cual una red neuronal entrenada correctamente puede clasificar con éxito los datos asociados a este problema de clasificación binaria. Sean folios de color rojo y azul que representan las clases del problema. Si se juntan dos folios de colores distintos y se hace una bola de papel con ellos se hace complicado separarlos uno del otro. Esta bola de papel representa los datos de entrada de la red. La información de cuál es cada uno de los folios sigue ahí pero es complicado distinguirlos en esa forma al igual que es difícil clasificar datos basándose en las variables de entrada únicamente. Las operaciones tensoriales que realiza una red neuronal pueden interpretarse como la secuencia de manipulaciones simples sobre la bola de papel para deshacerla y volver a obtener los folios iniciales aunque con ciertas arrugas que representarían las variables de salida de la red. Las arrugas en este caso representan las diferencias que existen entre la salida real y la salida obtenida. Está claro que una vez se deshace la bola de papel distinguir cada uno de los folios es mucho más sencillo.

En resumen una red neuronal realiza las operaciones tensoriales correspondientes para transformar geométricamente los datos de entrada, que pueden ser muy complejos, en una representación más sencilla que permita realizar la clasificación.

3. Aprendizaje

En el capítulo anterior se comentaron en profundidad las características de una red neuronal en términos de su arquitectura. En este sentido uno de los resultados más importantes era el hecho de que bajo ciertas condiciones la red neuronal era un aproximador universal. Más concretamente esto significaba que existía una configuración de pesos sinápticos óptima con la que la red reproducía la forma funcional adecuada para resolver cada problema concreto. Este capítulo centrará sus esfuerzos en describir y explicar cómo pueden conseguirse estos valores.

3.1. Concepto de Entrenamiento

En general en un modelo de aprendizaje automático el proceso por el cual se calculan los valores óptimos para los parámetros internos del modelo se conoce habitualmente como **aprendizaje** o **entrenamiento**. Nótese que ambas nomenclaturas revelan mucha información sobre la naturaleza del proceso.

En primer lugar el término aprendizaje sugiere que existe una cierta información a partir de la cual el modelo 'aprende', es decir, que adquiere la capacidad para reproducir el comportamiento asociado a esta información. En este caso la información corresponde a los valores numéricos de las variables de los conocidos como **datos de entrenamiento**. Por su parte el término entrenamiento deja entrever que existe un determinado proceso repetitivo que mejora el rendimiento del modelo de forma progresiva. Este proceso es lo que se conoce normalmente como **método de aprendizaje**.

En resumidas cuentas el entrenamiento de una red neuronal es un proceso que consiste en averiguar los valores de los pesos sinápticos que mejor reproducen el comportamiento de unos datos de entrenamiento usando para ello un determinado método de aprendizaje. Es importante destacar que los pesos sinápticos obtenidos dependen en gran medida de la calidad de los datos de entrenamiento. Si los datos de entrenamiento están sesgados por ejemplo, la red neuronal no será capaz de generalizar correctamente para nuevos datos.

3.2. Función de Coste

Para posibilitar el aprendizaje de la red es necesario conseguir la configuración de pesos que mejor reproduce el comportamiento de los datos de entrenamiento. La idea detrás de la afirmación anterior es que se produzca el mayor grado de coincidencia posible entre las variables objetivo o 'target' de los datos de entrenamiento t_j y las variables de salida calculadas por la red neuronal $y_j^{(l)}$.

Como es lógico, para cuantificar numéricamente este grado de coincidencia es necesario algún tipo de criterio que relacione las cantidades anteriores. Para ello se define lo que se conoce como **función de costo**. Como su propio nombre sugiere cuanto mayor sea el valor de la función de costo para una determinada configuración de pesos sinápticos

menor será la coincidencia entre las variables que buscamos.

En realidad la función de costo no es única, de hecho los investigadores han descubierto diferentes funciones de costo que pueden resultar más útiles según el tipo tarea con la que se trabaje. A continuación se describen dos funciones de costo muy utilizadas:

- **Error Cuadrático Medio.**

$$E = \frac{1}{N}\sum_{n=1}^{N}\left[\sum_{j=1}(y_j^{(l)} - t_j)^2\right]_n$$

- **Entropía Cruzada Binaria**

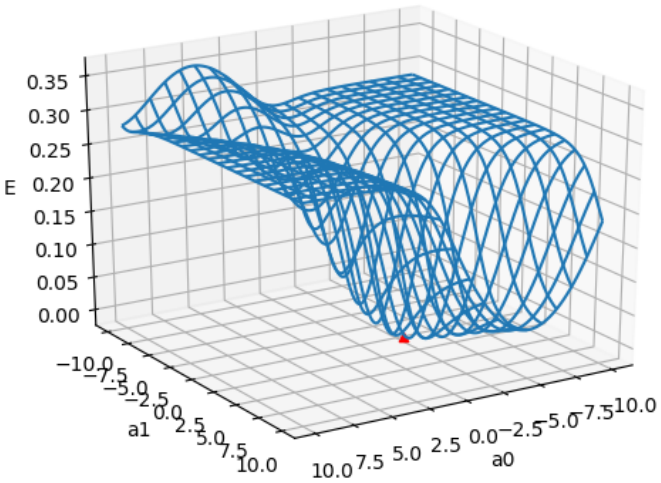
$$E = -\frac{1}{N}\sum_{n=1}^{N}\left[\sum_{j=1} t_j \log y_j^{(l)} + (1 - t_j)\log(1 - y_j^{(l)})\right]_n$$

Donde el subíndice n hace referencia a cada una de las entradas de los datos de entrenamiento siendo estas en total N. Además el índice j está sumado sobre todas las posibilidades.

Conceptualmente y conforme a lo que se verá en secciones venideras es muy interesante dar una interpretación geométrica de la función de costo. En primer lugar nótese que en el proceso de entrenamiento los parámetros variables serán los pesos sinápticos de modo que la función de costo puede expresarse en función de dichos pesos ya que los datos de entrenamiento serán los mismos.

Sea entonces una red neuronal que dispone de una única neurona y que pretende utilizarse para resolver un problema de regresión con ciertos datos de entrenamiento que disponen de una única variables de entrada. De acuerdo con la configuración anterior se deduce que se tienen dos pesos sinápticos distintos que denotaremos por a_0 y a_1. Si ahora se representa una función de costo determinada frente a los pesos sinápticos en el espacio tridimensional se obtendrá una **superficie de error** como la que se observa en la figura 5.

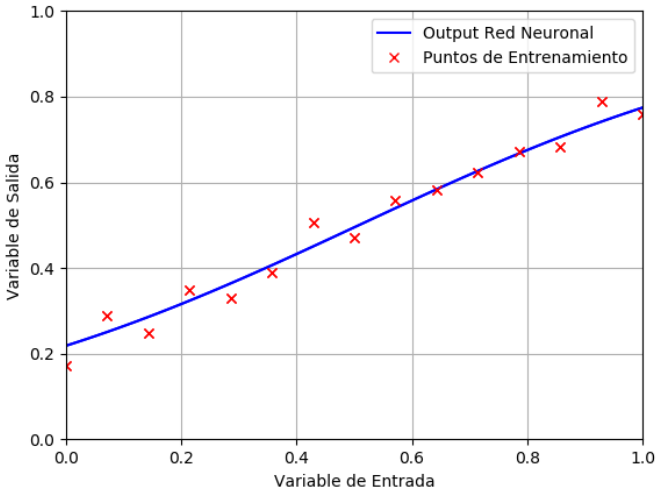
Figura 5: Superficie de Error.



La interpretación está clara, cuanto más elevados están los puntos en la superficie de error mayor es la discrepancia entre los datos de entrenamiento y la salida de la red para los pesos considerados. Por lo tanto si se quiere obtener la configuración de pesos óptima lo que debe hacerse es encontrar el mínimo sobre la superficie de error.

En la figura 6 se observa la comparación de los datos de entrenamiento y la salida de la red para la configuración de pesos que genera el mínimo sobre la superficie de error. Este mínimo es precisamente el triángulo rojo de la figura 5.

Figura 6: Resultados Regresión.



El concepto matemático de mínimo es fácilmente generalizable a un espacio de dimensión superior aunque no puede visualizarse gráficamente. La conclusión es que el entrenamiento de una red neuronal con m pesos sinápticos pasa por encontrar la configuración de pesos para la cual se obtiene el mínimo sobre la hipersuperficie de error en el espacio de dimensión $m+1$.

3.3. Método de Aprendizaje

Para encontrar el mínimo sobre la hipersuperficie de error que se viene de comentar pueden adoptarse diferentes estrategias, son lo que se conoce como **métodos de aprendizaje**.

3.3.1. Justificación

De acuerdo con lo comentado en la sección anterior uno podría pensar que el método más sencillo para encontrar el mínimo sobre la hipersuperficie de error es aplicar un enfoque basado en la **fuerza bruta**. La idea de este método es muestrear el espacio de posibles valores para cada uno de los pesos sinápticos de la red y encontrar un mínimo aproximado. Para conseguirlo se calcula el error para cada una de las configuraciones de pesos y se escogen los pesos que producen el menor error. A continuación se verá que este método es poco eficiente computacionalmente hablando motivo por el cual es necesario encontrar otra estrategia.

Para ver que el método por fuerza bruta es poco eficiente lo que hay que considerar es el número de configuraciones de pesos posibles ya que de este número dependerá la cantidad de veces que será necesario calcular la salida de la red neuronal para a su vez calcular el error cometido. Obviamente el número de configuraciones es en principio infinito pero puede hacerse un muestreo del espacio de pesos con una cierta granularidad m. La granularidad no es más que el número de configuraciones que se toma para cada uno de los pesos. Definiendo g como el número de configuraciones a calcular y p el número de pesos de la red neuronal es obvio que $g = m^p$. Supóngase ahora una red con l capas y

un vector $arch_k$ de dimensión $l+1$ con $k = 0, ..., l$ que representa la arquitectura de la red donde $arch_0$ es el número de variables de entrada y los elementos restantes son el número de nodos en cada una de las capas de la red. Considerando la notación anterior y suponiendo que se emplea un 'bias term' como peso adicional puede comprobarse que el número de configuraciones g puede expresarse como:

$$\log_m g = \sum_{k=1}^{l} arch_k(arch_{k-1} + 1)$$

A continuación se muestran dos ejemplos asumiendo una granularidad $m = 5$ para ver que el número de configuraciones escala muy rápidamente:

- **Ejemplo 1.** Supóngase una red de una única capa $l = 1$ con tres variables de entrada y una única variable de salida, es decir, $arch = (3, 1)$. En virtud de la fórmula anterior el número de configuraciones que habría que probar es:

$$g = 5^4 = 625$$

- **Ejemplo 2.** Supóngase ahora el problema anterior con las mismas variables de entrada y salida pero con una red de dos capas $l = 2$ teniendo la capa oculta 6 nodos. En ese caso $arch = (3, 6, 1)$ y el número de configuraciones que habría que probar es ahora:

$$g = 5^{31} \approx 4.66 \times 10^{21}$$

La conclusión es clara, el primer ejemplo sería fácilmente computable mientras que el segundo sería imposible. Obviamente usar un m mayor no haría más que empeorar las cosas. Por lo tanto se demuestra que el método basado en la fuerza bruta no es suficiente para resolver el problema del entrenamiento con toda generalidad.

3.4. Método del Gradiente Descendente

En el campo de las matemáticas existen cantidad de algoritmos que permiten encontrar el mínimo de una función en un tiempo computacional razonable pero en este caso se utilizará el llamado **método del gradiente descendente**. Este suele aplicarse habitualmente en diversos problemas de 'machine learning' y en este caso podría utilizarse para encontrar la configuración de pesos sinápticos que minimiza la función de costo.

En primer lugar es necesario comentar que el método del gradiente descendente es un método numérico, es decir, las variables correspondientes al mínimo serán obtenidas de manera aproximada. El principio básico que se utiliza para conseguir el mínimo proviene de la propia definición del gradiente. En líneas generales el método del gradiente descendente parte de una configuración inicial de variables y va modificando cada una de ellas incrementándola o disminuyéndola en la dirección contraria a la del gradiente de la

función que pretende minimizarse. Se trata además de un proceso iterativo en el que se repite la misma operación un número fijo de iteraciones finalizando con la obtención de la estimación del valor del mínimo. En el caso del problema del aprendizaje en las redes neuronales las variables que se van modificando son los pesos sinápticos mientras que la función a minimizar sería la función de costo que de alguna forma mide el error que comete la red.

Cabe destacar que existen múltiples variaciones o alternativas que pueden utilizarse en el procedimiento anterior con el objetivo de solventar de la mejor forma algunos problemas que pueden surgir durante la obtención del mínimo. Tal vez el problema más relevante es la obtención de una estimación correspondiente a un mínimo local de la función a minimizar. Más adelante se discutirán con más detalle algunas técnicas para hacer que el proceso de aprendizaje sea más eficiente.

Para entender mejor todo lo anterior y dado que el gradiente descendente tiene sus raíces en la teoría del cálculo diferencial es conveniente dar una interpretación geométrica del problema. Recuérdese que desde el punto de vista geométrico el problema consiste en encontrar el punto en el espacio de valores para los pesos sinápticos correspondiente al mínimo de la hipersuperficie de error dada por la función de costo. Suponiendo una superficie de error n-dimensional asociada a n pesos sinápticos, el gradiente de la función de costo en un punto arbitrario indica la dirección del espacio n-dimensional en la que el valor de dicha función aumenta de manera más acusada, es decir, con más pendiente. Este gradiente permite saber en qué dirección es necesario mover los pesos sinápticos para minimizar el error, esta es exactamente la dirección opuesta al gradiente. Cuando el valor de los pesos se modifica el gradiente en dicho punto cambia de modo que debe adoptarse una nueva dirección en la que mover los pesos justificando así la necesidad de iterar repetitivamente la operación.

3.4.1. Propagación hacia Atrás del Error

Tal y como se acaba de ver, el pilar fundamental del método del gradiente descendente es el gradiente de la función de costo. Por ende será indispensable conocerlo explícitamente para poder hallar el mínimo deseado.

Nótese que la función de costo es una función que en última instancia depende de los pesos sinápticos utilizados en la red:

$$E = E(w_{ij}^{(k)})$$

En tal caso hallar el gradiente de la función anterior es equivalente a encontrar todas y cada una de las derivadas parciales de la forma: $\frac{\partial E}{\partial w_{ij}^{(k)}}$

Para una red neuronal simple con pocas capas calcular las derivadas parciales anteriores es relativamente simple haciendo uso de la **regla de la cadena** sin embargo cuando se tienen redes más complicadas con más de una capa oculta el cálculo resulta más farragoso

y se hace bastante difícil implementar un algoritmo que realice el cálculo con toda generalidad. Para resolver este problema surge el **algoritmo de propagación hacia atrás**. A continuación se detalla el funcionamiento concreto del algoritmo utilizando la misma notación que la usada en el capítulo 1.

En primer lugar a cada una de las derivadas parciales buscadas se le asocia el valor:

$$\frac{\partial E}{\partial w_{ij}^{(k)}} = \delta_j^{(k)} \frac{\partial u_j^{(k)}}{\partial w_{ij}^{(k)}}$$

La función $\delta_j^{(k)}$ (no confundir con la delta de Dirac) es una función que puede asociarse a cada uno de los nodos j de la capa k correspondiente de la red y que permite simplificar la notación notablemente agrupando el valor de varias derivadas parciales que van apareciendo en el cálculo. Sin embargo, la mayor ventaja que ofrece la utilización de $\delta_j^{(k)}$ es que puede calcularse iterativamente para cada capa k sabiendo cual es su valor en la capa siguiente $k+1$, facilitando así enormemente la implementación de su cálculo en un lenguaje de programación. Más concretamente $\delta_j^{(k)}$ puede calcularse a partir de $\delta_j^{(k+1)}$ a través de la expresión:

$$\delta_j^{(k)} = \sum_{m=1} \delta_m^{(k+1)} \sum_{n=1} w_{nm}^{(k+1)} \frac{\partial y_n^{(k)}}{\partial u_j^{(k)}}$$

Donde las sumas anteriores se realizan sobre todos los índices posibles. Esta expresión es la que le da su nombre al algoritmo de propagación hacia atrás porque lo que se está haciendo esencialmente es calcular primero las derivadas parciales del error en la capa más profunda de la red y a continuación 'propagar' estas derivadas del error hacia atrás en la red, es decir, a capas anteriores, para poder conocer las derivadas del error restantes.

Nótese que para poder calcular $\delta_j^{(k)}$ para todos y cada uno de los k posibles es necesario conocer de algún otro modo $\delta_j^{(l)}$ donde l representa la última capa de la red. Puede demostarse con cierta facilidad usando la regla de la cadena que el valor de $\delta_j^{(l)}$ es:

$$\delta_j^{(l)} = \sum_{m=1} \frac{\partial E}{\partial y_m^{(l)}} \frac{\partial y_m^{(l)}}{\partial u_j^{(l)}}$$

Adicionalmente puede deducirse fácilmente en base a la expresión $u_j^{(k)} = w_{0j}^{(k)} + \sum_{i=1}^{n} w_{ij}^{(k)} y_i^{(k-1)}$ del capítulo 1 que:

$$\frac{\partial u_j^{(k)}}{\partial w_{ij}^{(k)}} = \begin{cases} 1 & \text{si } i = 0 \\ y_i^{(k-1)} & \text{si } i \neq 0 \end{cases}$$

De esta forma quedan totalmente determinadas cada una de las derivadas parciales necesarias para utilizar el método del gradiente descendente. Nótese que las expresiones

anteriores son totalmente generales y que se simplifican notablemente con las condiciones adecuadas, por ejemplo si la función de activación actúa de manera independiente en cada uno de los nodos entonces se cumple $\frac{\partial y_m^{(k)}}{\partial u_j^{(k)}} \neq 0$ si y sólo si $m = j$.

3.4.2. Reglas del Aprendizaje

En la sección anterior se ha descrito en detalle cómo calcular las distintas derivadas del error dado por la función de costo con respecto a cada uno de los pesos de la red. Tal y como establece el método del gradiente descendente la magnitud y dirección en la que debemos modificar los pesos de la red dependen de estas derivadas parciales sin embargo existen diferentes criterios que pueden seguirse para realizar la modificación. Los métodos o criterios que se utilizan para calcular cuánto cambia el peso se conocen como **reglas del aprendizaje**.

Téngase en cuenta que el proceso de aprendizaje se lleva a cabo de manera iterativa un determinado número de veces para alcanzar la configuración óptima de pesos. Cada una de estas iteraciones recibe el nombre de **época** de modo que a partir de aquí y únicamente en esta sección se utilizará w_m para referirse al valor de un peso arbitrario $w_{ij}^{(k)}$ en una determinada época m. Adicionalmente se asumirá, si no se especifica lo contrario, que cualquier cantidad con el subíndice m es en principio diferente para cada peso de la red. Con esta notación el cambio de un peso arbitrario de una época a la siguiente puede expresarse matemáticamente como:

$$w_m = w_{m-1} + c_m$$

Donde c_m denota el cambio que se realiza en el peso en una época determinada. Nótese que las reglas de aprendizaje esencialmente sirven para especificar la cantidad y el sentido en el que deben modificarse los pesos, información que en última instancia determina el valor de c_m.

Existen muchas reglas distintas y cada una de ellas responde a cierto tipo de problemática concreta. En este caso se centrará la mirada sobre un criterio bastante común conocido como regla de la **tasa de aprendizaje adaptativa**.

Sea un caso general en el que se dispone de N entradas o datos para evaluar con la red y uno se encuentra en la época m del proceso de aprendizaje. El primer paso del método es calcular las derivadas parciales vistas en la sección anterior para cada una de las entradas y hacer un agregado de las mismas referente a cada peso. Esto da un parámetro d_m para cada peso dado por:

$$d_m = \sum_{n=1}^{N} \left(\frac{\partial E}{\partial w_m} \right)_n$$

Dado que el parámetro anterior contiene la información de las derivadas uno podría

decir que en primera aproximación $c_m \approx -d_m$ dado que se deberían mover los pesos en la dirección opuesta a la que indica el gradiente del error. Esto es correcto en parte pero si sólamente se hiciese esto encontrar la configuración de pesos con el mínimo de error sería un proceso muy ineficiente y en algunos casos casi imposible.

Generalmente todas las reglas de aprendizaje introducen un factor (generalmente constante) ϵ_m denominado **tasa de aprendizaje** que modula la cantidad d_m para conseguir un aprendizaje más eficiente. La aportación original de la regla de la tasa de aprendizaje adaptativa consiste en hacer que cada uno de los pesos tenga una tasa de aprendizaje particular y que esta vaya cambiando en cada época de forma independiente según el criterio siguiente. Dada una época, si la dirección en la que debe moverse un peso concreto para que el error disminuya es la misma que la dirección en la que debía moverse recientemente, entonces se incrementa el valor de la tasa de aprendizaje asociada a ese peso. Si por el contrario la dirección en la que debe moverse el peso es la opuesta a la dirección en la que debía moverse en un pasado reciente, entonces se reduce el valor de la tasa de aprendizaje.

La dirección en la que se movieron los pesos 'recientemente' puede definirse matemáticamente a través de un promedio f_m de la forma: $f_m = \theta f_{m-1} + (1-\theta)d_{m-1}$. Donde $0 \leq \theta \leq 1$ es un parámetro fijado por el usuario que controla cómo de largo es el 'reciente' utilizado. La interpretación del signo de f_m es exactamente la misma que la de d_m.

Una vez se conocen el d_m y el f_m de un peso, averiguar cuánto se debe modificar la tasa de aprendizaje ϵ_m según el criterio explicado con anterioridad puede especificarse mediante la expresión:

$$\epsilon_m = \begin{cases} \epsilon_{m-1} + \kappa & \text{si } d_m f_m > 0 \\ \epsilon_{m-1} \cdot \phi & \text{si } d_m f_m < 0 \end{cases}$$

Donde κ y $0 \leq \phi \leq 1$ son también parámetros fijados por el usuario que regulan cómo de grandes son los cambios sobre la tasa de aprendizaje.

Con todo lo explicado hasta ahora el cambio en los pesos sería simplemente $c_m = -\epsilon_m d_m$ pero pueden introducirse opcionalmente dos variantes que tienen ciertas ventajas: el **momento** en el aprendizaje y el **aprendizaje por lotes**.

El aprendizaje por lotes consiste en calcular el parámetro d_m utilizando una pequeña parte o lote del conjunto de datos de entrenamiento en lugar de usar la totalidad del mismo. En otras palabras los pesos se modificarían más de una vez en cada época en función de lo grande o pequeño que sea el lote usado. El caso límite sería cambiar los pesos cada vez que se procesa una entrada de los datos de entrenamiento, es decir, aprendizaje ejemplo a ejemplo.

Añadir un momento en el aprendizaje consiste en hacer que el cambio de los pesos en una época determinada sea un promedio del cambio que correspondería habitualmente a ese peso y el cambio que sufrió en la época anterior. Matemáticacmente esto puede expresarse como: $c_m = \mu c_{m-1} - (1-\mu)\epsilon_m d_m$. Donde el parámetro $0 \leq \mu < 1$ fijado por el usuario es el que controla cómo de importante es cada término en el promedio.

4. Lista de Algoritmos

Téngase en cuenta que en los siguientes algoritmos aparecen funciones como ACTIVATION, DERIV_COST y DERIV_ACTIV que representan la acción de la función de activación, la derivada de la función de coste y la derivada de la función de activación. Naturalmente su valor dependerá de la función de activación concreta y de la función de costo utilizada. Finalmente comentar que la variable "activación" es una lista que contiene la función de activación utilizada en cada capa por lo tanto también es relevante en las funciones ACTIVATION y DERIV_ACTIV.

Algorithm 1 Entrenamiento de la Red Neuronal

```
1:  procedure TRAIN(x, t, epochs, batchsize, activation, costfunction)
2:      for m ← 0 to epochs − 1 do
3:          for n ← 0 to LENGTH(x) − 1 do
4:              y ← MAPPING(x, n, w, activation)
5:              d ← BACKPROPAGATION(y, t, d, activation, costfunction)
6:              if n MOD batchsize = 0 then
7:                  w, d, f, c, epsilon ← CHANGEWEIGHT(w,d,f,c,epsilon,kappa,phi,theta,mu)
8:              end if
9:          end for
10:     end for
11: end procedure
```

Algorithm 2 Mapeado de la Red Neuronal

```
1:  procedure MAPPING(x, n, w, activation)
2:      set of var y ← ∅
3:      y[0] ← x[n]
4:      for k ← 0 to LENGTH(w) − 1 do                ▷ Iteración sobre las Capas
5:          for j ← 0 to LENGTH(w[k][0]) − 1 do      ▷ Iteración sobre los Nodos
6:              y[k + 1][j] ← w[k][0][j]
7:              for i ← 0 to LENGTH(y[k]) − 1 do     ▷ Iteración sobre las Variables
8:                  y[k + 1][j] = y[k + 1][j] + w[k][i][j] · y[k][i]
9:              end for
10:         end for
11:         y[k + 1] = ACTIVATION(y[k + 1], activation[k])
12:     end for
13: end procedure
```

Algorithm 3 Propagación Hacia Atrás del Error

procedure BACKPROPAGATION(y, t, d, $activation$, $costfunction$)
  set of var $delta \leftarrow \emptyset$; **set of var** $deriv \leftarrow \emptyset$; $layers = \text{LENGTH}(w)$
  $delta[layers] = \text{DERIV\_COST}(y[layers], t) \cdot \text{DERIV\_ACTIV}(y[layers])$
  $deriv[layers][0] = delta[layers]$
  for $i \leftarrow 0$ **to** $\text{LENGTH}(y[layers - 1]) - 1$ **do**
    $deriv[layers][i + 1] = delta[layers][i + 1] \cdot y[layers - 1][i]$
  end for
  $d[layers] = d[layers] + deriv[layers]$
  for $k \leftarrow 1$ **to** $layers - 1$ **do**
    $delta[layers - k - 1] = \text{SUM}(delta[layers - k] \cdot w[layers - k][:]) \cdot \text{DERIV\_ACTIV}(y[layers - k - 1])$
    $deriv[layers - k - 1][0] = delta[layers - k - 1]$
    for $i \leftarrow 0$ **to** $\text{LENGTH}(y[layers - k - 2]) - 1$ **do**
      $deriv[layers - k - 1][i + 1] = delta[layers - k - 1] \cdot y[layers - k - 2][i]$
    end for
    $d[layers - k - 1] = d[layers - k - 1] + deriv[layers - k - 1]$
  end for
end procedure

Algorithm 4 Cambio de Pesos Sinápticos

procedure CHANGEWEIGHT(w, d, f, c, $epsilon$, $kappa$, phi, $theta$, mu)
  for $k \leftarrow 0$ **to** $\text{LENGTH}(w) - 1$ **do** ▷ Iteración sobre las Capas
    for $j \leftarrow 0$ **to** $\text{LENGTH}(w[k][0]) - 1$ **do** ▷ Iteración sobre los Nodos
      for $i \leftarrow 0$ **to** $\text{LENGTH}(w[k]) - 2$ **do** ▷ Iteración sobre las Variables
        if $d[k][i][j] \cdot f[k][i][j] > 0$ **then**
          $epsilon[k][i][j] \leftarrow epsilon[k][i][j] + kappa$
        end if
        if $d[k][i][j] \cdot f[k][i][j] \leq 0$ **then**
          $epsilon[k][i][j] \leftarrow epsilon[k][i][j] \cdot phi$
        end if
        $f[k][i][j] \leftarrow (1 - theta) \cdot d[k][i][j] + theta \cdot f[k][i][j]$
        $c[k][i][j] \leftarrow (1 - mu) \cdot (-1) \cdot e[k][i][j] \cdot d[k][i][j] + mu \cdot c[k][i][j]$
        $w[k][i][j] \leftarrow w[k][i][j] + c[k][i][j]$
        $d[k][i][j] \leftarrow 0$ ▷ Reseteamos el valor de las derivadas
      end for
    end for
  end for
end procedure

5. Aplicación Red Neuronal

5.1. Descripción de la Aplicación

En esta sección se pretende describir una aplicación que permite utilizar una red neuronal genérica definida por el usuario. Adicionalmente la aplicación presenta otras funcionalidades de interés que se verán posteriormente. Comentar que la implementación de la aplicación se ha realizado utilizando el lenguaje Python y el código asociado a la misma puede encontrarse en un repositorio de GitHub con la dirección `https://github.com/josedamiangarcianoya/tfg_neural_network`

El elemento principal de la aplicación es un **objeto** denominado **ANN** (Artificial Neural Network) con el que pueden realizarse todas y cada una de las funcionalidades disponibles.

El primer paso para utilizar una red neuronal es definirla, acción para la que es necesario llamar al objeto ANN() y especificar cuatro variables con las que la red queda totalmente determinada. Estas variables son: el número de nodos de entrada con el que se va a trabajar (**inputs**), la tarea que se pretende realizar bien regresión o bien clasificación (**task**), una lista (**arch**) con tantos elementos como capas tenga la red donde cada elemento representa el número de nodos de la capa correspondiente y finalmente otra lista (**activation**) también con tantos elementos como capas donde cada elemento es una cadena de caracteres que identifica la función de activación que se desea utilizar en la capa correspondiente.

Esta versión de la aplicación consta de tres funciones de activación disponibles: reLU, logística y softmax. Estas son más que suficientes para tareas simples aunque la implementación se ha hecho de manera suficientemente general como para poder incluir otras funciones de activación en el programa simplemente especificando un indificador y la definición de la propia función.

La primera función de interés es **add_layer** que permite añadir una capa adicional a una red definida con anterioridad. Como es lógico, es necesario especificar el número de nodos de la capa y la función de activación utilizada.

Otra función interesante es **preprocessing** que permite realizar como su propio nombre indica un preprocesado sobre unos datos de entrada. Entre otras cosas se incluye una normalización de los valores numéricos de entrada y una conversión de los datos categóricos en tantas variables binarias como clases se tengan.

La siguiente función **learning_options** es de alguna forma un precursor del entrenamiento de la red y permite especificar algunos parámetros de interés para el mismo. Entre ellos se encuentran la función de costo a utilizar y los parámetros κ, ϕ, θ y μ relativos al aprendizaje. Esta versión de la aplicación consta de dos funciones de activación disponibles: el error cuadrático medio y la entropía cruzada binaria.

Continuando con la tarea del entrenamiento de la red se dispone de la función **train** que requiere de los datos de entrenamiento, la salida esperada a estos datos, el número de

épocas a utilizar y el tamaño de los lotes si es que se van a utilizar.

Una vez que la red ha sido entrenada puede utilizarse la función **predict** para conocer a partir de unos datos de entrada la salida predicha por el modelo. En caso de haber realizado el preprocesado de unos datos categóricos utilizando la aplicación es posible elegir entre especificar la probabilidad predicha para cada clase o directamente dar la variable categórica más probable.

Finalmente se adjunta una función **evaluate** que permite realizar la tarea de evaluación del modelo a partir de un set completo de datos. Se deben especificar algunos parámetros adicionales como las épocas de entrenamiento y el número de divisiones k para hacer validación cruzada. Concretamente lo que hace esta función es dividir el set de datos de entrenamiento en k grupos del mismo tamaño aproximadamente. A continuación $k-1$ grupos se utilizan como entrenamiento de la red y el grupo restante se utiliza para testear el modelo por medio de una métrica que evalua el rendimiento. El proceso se repite k veces variando el grupo utilizado para testeo cada vez. Finalmente se hace una media de los rendimientos obtenidos para cada grupo para obtener el rendimiento final del modelo.

En cuanto a las métricas utilizadas para testear el modelo se han implementado el **MAE** (Mean Absolute Error) para regresión y la **Accuracy** o Precisión para clasificación.

$$MAE = \sum_{i=1}^{n} \frac{|y_i - t_i|}{n} \qquad Accuracy = \frac{\text{Predicciones Correctas}}{\text{Predicciones Totales}}$$

5.2. Comparativa Aplicación Vs. Keras

Para finalizar este trabajo se ofrece una comparativa del rendimiento obtenido para una red implementada por medio de la aplicación descrita anteriormente frente a otra red de la misma arquitectura implementada mediante la famosa librería de aprendizaje automático, Keras. Así mismo se realizó una camparativa para problemas clásicos de regresión y clasificación como pueden ser 'Boston Housing Dataset' y 'Iris Flower Dataset' respectivamente.

Los rendimientos obtenidos fueron:

Boston Housing	**Iris Flower**
$MAE_{(App)} = 0{,}0639$	$Accuracy_{(App)} = 0{,}9662$
$MAE_{(Keras)} = 0{,}0971$	$Accuracy_{(Keras)} = 0{,}9257$

Bibliografía

[1] Murray Smith, "Neural Networks for Statistical Modeling", January 1993.

[2] Ke-Lin Du, M.N.S. Swamy, "Neural Networks and Statistical Learning", Springer London 2014.

[3] F. Chollet, "Deep Learning with Python", Manning Publications 2018

[4] I. Vasilev, D. Slater et al., "Python Deep Learning", Packt Publishing Ltd. 2019.

[5] Dusty Phillips, "Python 3 Object Oriented Programming", Packt Publishing 2010.

[6] G. Cybenko, "Aproximation by Superposition of a Sigmoidal Function", December 1989.

[7] K. Hornik, "Multilayer Feedforward Networks are Universal Aproximators", March 1989.

[8] Liliana Teodorescu, "Artifical Neural Networks in High-Energy Physics", March 2006.

[9] Angelo Ciaramella et al., "Applications of Neural Networks in Astronomy and Astroparticle Physics", 2005.