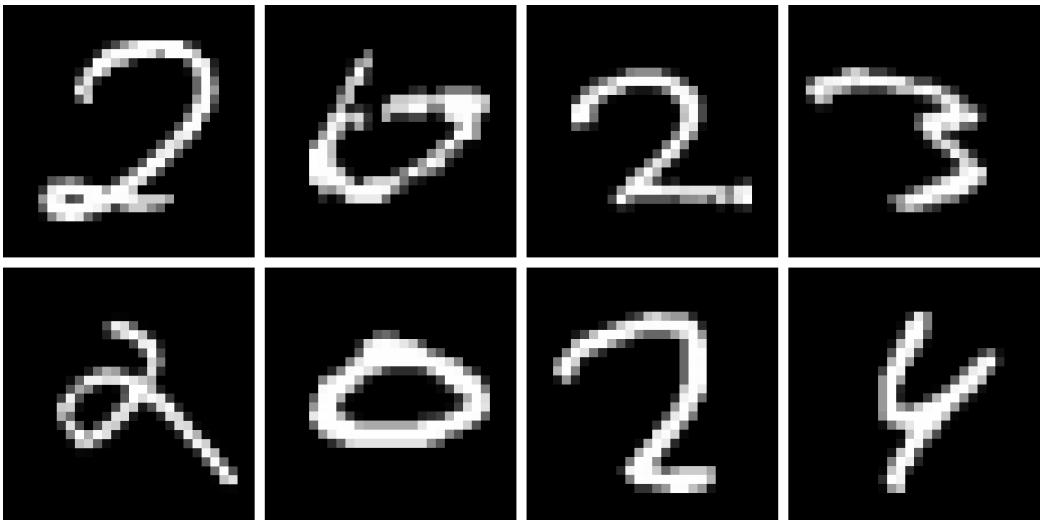
¿Qué hay detrás de una red neuronal?

Russell



Tecnología
2º Bachillerato
Curso 2023 - 2024

Una vida sin reflexión no merece ser vivida

- Sócrates

It seems that artificial intelligence is becoming more and more part of our everyday lives. What apparently came from nowhere is currently on its way to dominate on virtually every field imaginable. Despite this, when I started this project, I knew very little about how these technologies actually worked. The main objective of the paper is to solve this problem, and understand, at least on a smaller scale, what makes artificial intelligence, intelligent

During this work, we will mainly look into neural networks, as they serve a pretty good foundation for the more general concept of artificial intelligence. In the theoretical framework, we dive into the mathematics of a neural network. From how it interprets data, to how it learns. We will look into concepts like gradient descent and backpropagation, which is how the network actually learns.

Once we determine how a neural network manages to learn, we will apply this new-found knowledge into creating our own neural network. Written in Python, our network will be able to identify handwritten digits. At first, this may seem trivial for a human to do such a task, but it becomes increasingly difficult to tell a computer how to identify the edges that form a number. Hence, making it a perfect case to learn about neural networks.

Using the MNIST database, our neural network will be able to identify handwritten digits with a 92 % accuracy, which admittedly is not very good when compared to other neural networks, although as a first attempt at making one, it is still a great result.

Índice

1. Introducción **7**

2. Marco Teórico **9**
2.1. Historia de la inteligencia artificial 9
2.2. Diferencias entre inteligencia artificial, *machine learning* y redes neuronales 9
2.2.1. Inteligencia artificial 9
2.2.2. Machine learning 10
2.2.3. Redes Neuronales 11
2.3. Estructura de una red neuronal 13
2.3.1. Nodos 14
2.3.2. Interacciones entre capas 15
2.3.3. Ejemplo de la estructura 20
2.4. ¿Cómo aprende una red neuronal? 24
2.4.1. Coste 26
2.4.2. Descenso de Gradiente del coste 28
2.4.2.1. Descenso de gradiente aplicado a 2 dimensiones: 28
2.4.2.2. Descenso de gradiente aplicado a 3 dimensiones: 31
2.4.3. Retropropagación 37
2.4.3.1. Regla de la cadena 38
2.4.3.2. Regla de la cadena aplicada a multiples nodos 44

3. Marco Práctico **51**
3.1. Objetivos de nuestra red neuronal 51
3.2. Hagamos una red neuronal 52
3.2.1. Programación Orientada a Objetos 52
3.2.2. Estructura de nuestra red neuronal 54
3.2.2.1. Clase Layer 54
3.2.2.2. Clase Network 58
3.2.3. main.py 66
3.2.4. test.py 75
3.3. Resultados 80

4. Conclusión **82**

5. Bibliografía **83**

6. Anexo **85**
6.1. Layer.py 85
6.2. Network.py 85
6.3. main.py 90
6.4. test.py 93

1. Introducción

En la actualidad, las redes neuronales y la inteligencia artificial han alcanzado niveles de sofisticación y capacidad que parecen sacados directamente de la ciencia ficción. Ejemplos emblemáticos como DALL-E 2, la última iteración de un modelo generativo capaz de crear imágenes y obras de arte a partir de descripciones textuales, o ChatGPT, un sistema de procesamiento de lenguaje natural que permite mantener conversaciones profundas y significativas, demuestran el asombroso avance en el campo de la inteligencia artificial. Estas tecnologías representan un punto de inflexión en la forma en que interactuamos con la información y, en última instancia, transforman la forma en que trabajamos y nos relacionamos con el mundo que nos rodea.

Estos avances son cada vez más impresionantes, hasta el punto donde un humano puede tener dificultades para distinguir entre algo humano o creado por una inteligencia artificial. Por ejemplo, el parágrafo anterior ha sido escrito por una inteligencia artificial

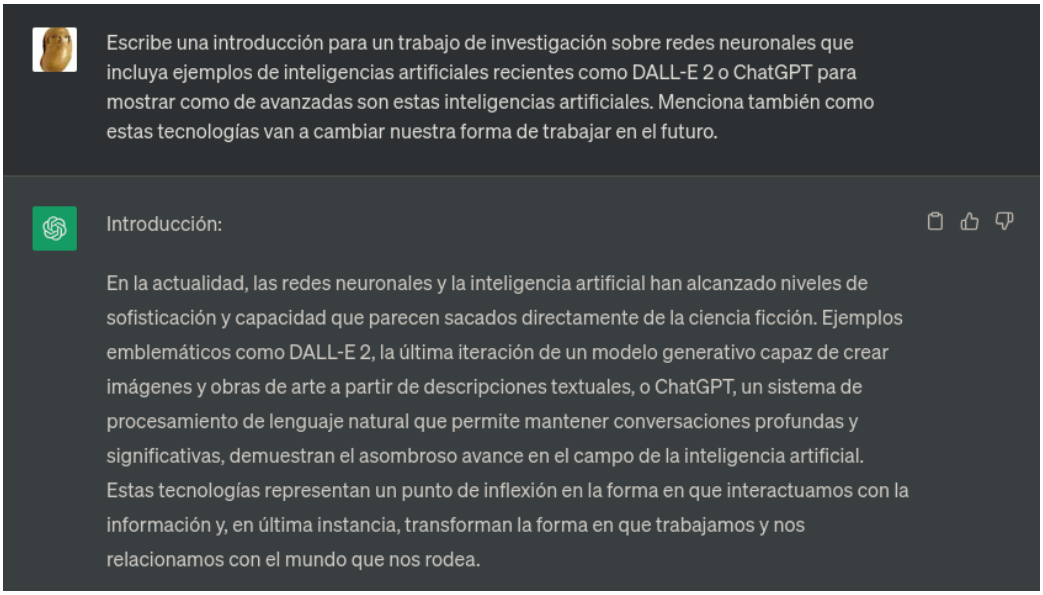


Figura 1: ChatGPT redactando el primer párrafo de la introducción

Claramente, la inteligencia artificial ha llegado para quedarse, y sin duda va a revolucionar nuestra forma de trabajar en el futuro. Por esto pienso, que es importante tener un conocimiento básico sobre cómo funcionan estas inteligencias artificiales, y es este el principal objetivo de este trabajo.

Este trabajo se centrará en los fundamentos de estas tecnologías, sin entrar en detalles técnicos avanzados, por su alto grado de complejidad. Se

explicará la estructura fundamental que compone una red neuronal, se verá cómo puede llegar a “aprender” una inteligencia artificial, y finalmente se intentará aplicar este conocimiento para crear una inteligencia artificial.

2. Marco Teórico

2.1. Historia de la inteligencia artificial

El concepto de inteligencia artificial se ha discutido desde hace siglos, en obras como Frankenstein por Mary Shelley, o si nos vamos aún más atrás, podemos encontrar en la mitología griega una apelación a la inteligencia artificial con el autómata Talos. Sin embargo, estas referencas a la inteligencia artificial tratan el ámbito ético, que no es el objetivo de este trabajo.[1]

El concepto de que un razonamiento pueda estar definido por normas, será introducido por Aristóteles, con las normas del silogismo. Ramón Llull, seguirá por este camino, en su libro Ars Magna, donde propone que un razonamiento se puede hacer de forma artificial.[2]

En el ámbito tecnológico, Ada Lovelace, comúnmente considerada la primera programadora, especuló en 1840 que estas normas se podrían aplicar a un ordenador, y especuló que un ordenador podría llegar hasta a componer una canción por sí solo. Pero no será hasta el 1956 que se empieza a hablar de inteligencia artificial.[3]

En 1956, John McCarthy y Marvin Minsky, organizaron el Dartmouth Workshop, donde reunieron a múltiples matemáticos y científicos para desarrollar las bases de la inteligencia artificial. Este evento se contempla como el nacimiento de la inteligencia artificial que conocemos hoy.[4]

2.2. Diferencias entre inteligencia artificial, *machine learning* y redes neuronales

Antes de continuar con este trabajo es necesario aclarar los siguientes conceptos que comúnmente se usan como sinónimos, pero no lo son.

2.2.1. Inteligencia artificial

La inteligencia artificial es el estudio de máquinas y programas que imitan la inteligencia humana. Pero esta definición es muy amplia.

[1] *History of artificial intelligence*. Jun. de 2023. URL: `https://en.wikipedia.org/wiki/History_of_artificial_intelligence`.

[2] *Historia de la inteligencia artificial*. Jun. de 2023. URL: `https://es.wikipedia.org/wiki/Historia_de_la_inteligencia_artificial`.

[3] *Historia de la inteligencia artificial*.

[4] *History of artificial intelligence*.

Derivamos de esta definición 3 categorías de inteligencias artificiales:[5]

- ANI: *Artificial Narrow Intelligence* (Inteligencia artificial estrecha)
- AGI: *Artificial General Intelligence* (Inteligencia artificial general)
- ASI: *Artificial Super Intelligence* (Superinteligencia artificial)

La ANI es donde estamos con las inteligencias artificiales (IA) actuales. Las IA convencionales utilizadas para jugar al ajedrez, o determinar patrones, forman parte de esta categoría, dado que con los datos que se le presentan, la IA es capaz de aprender y devolver una respuesta.

La AGI y ASI supondría una IA capaz de resolver cualquier problema que un humano pueda entender, y en el caso de la ASI, superar la capacidad del humano.

La diferencia entre una ANI y una AGI, es que a AGI no se tiene que especializar. Una ANI puede ganar a cualquier jugador al ajedrez, porque es para lo que ha sido entrenada, pero si le pides jugar a las damas, o al tres en raya, o hacer cualquier cosa que no sea jugar al ajedrez, no podrá hacerlo, porque solo ha sido programada para jugar a ajedrez.

En comparación, una AGI, estaría programada para hacer una multitud de tares, de la misma forma que un humano puede jugar al ajedrez, pero también a las damas, o al póker, o a cualquier otra cosa. Inteligencias como ChatGPT que pueden hacer una multitud de cosas, se podrían considerar como el principio de las AGI, pero como estas definiciones no son muy técnicas, cuesta cuantificar su inteligencia.

Cabe destacar que estas categorías no son enfoques diferentes hacia una IA, sino que sirven para medir el nivel de inteligencia de las IA.

2.2.2. Machine learning

A la hora de programar una IA, existen diferentes enfoques, con objetivos diferentes. El objetivo principal en la actualidad es llegar a una AGI, que sea capaz de resolver los mismos problemas que un humano. Sin embargo, esto aún queda un poco fuera de nuestro alcance, y, por lo tanto, se desarrollan IA con objetivos más concretos.

Uno de estos enfoques sería el *machine learning* que destaca por ser capaz de "aprender" como los humanos según se le va dando información.

[5]Eda Kavlakoglu. *AI vs. Machine Learning vs. Deep Learning vs. neural networks: What's the difference?* Mayo de 2020. URL: `https://www.ibm.com/cloud/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks`.

A la IA se le hace "aprender" las relaciones entre diferentes datos, para que cuando se le enseñe datos nuevos, sea capaz de encontrar las mismas relaciones y obtener la respuesta deseada.[6]

Por ejemplo, a una IA de *machine learning* se le puede dar una colección de fotos con la edad de las personas fotografiadas. La IA entonces es capaz de encontrar relaciones entre la foto y la edad. Después, ante fotos nuevas, nunca antes mostradas a la IA, de las que no sabe la edad, es capaz de encontrar las mismas relaciones para determinar la edad de las fotos nuevas.

Este tipo de IA se usa, por ejemplo, en los motores de búsqueda de Google o Bing, los motores de recomendaciones de Netflix o YouTube, etc.

En Netflix, por ejemplo, cada vez que decides ver una película o una serie, la IA recibe estos datos, y aprende de ellos para que las próximas recomendaciones sean mejores. Si la IA ve que de las series que ha recomendado a un usuario, este selecciona una serie policiaca, aprende de esto y le recomendará más series policiacas.

2.2.3. Redes Neuronales

Una red neuronal, o *artificial neural network* (ANN)[7] es una manera de programar una IA con *machine learning*. Está basada en una estructura de nodos, y conexiones entre estos nodos, como se ve en la figura 2:

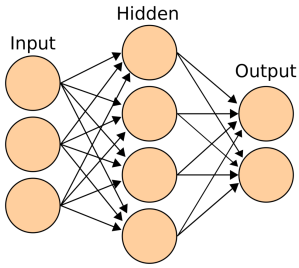


Figura 2: Diagrama de una red neuronal

Más adelante en el trabajo profundizaremos sobre la estructura de una red neuronal, pero por ahora debemos saber que está compuesta por los nodos naranjas, y estos están conectados con el resto de nodos.

En la capa de input, se establece un valor entre 1 y 0 para cada nodo. Este valor llega al los nodos de la siguiente capa, **pero es modificado un poco por la conexión, por algo que se conoce como peso**. Este nuevo valor llega a la capa de output, una vez más siendo modificada por los pesos. Y el valor que obtenemos en el output será nuestra respuesta deseada.

Vemos de esta pequeña explicación, qué son los pesos los que "toman" una decisión. Pero la complicación reside en encontrar esos pesos de forma que

[6] Kavlakoglu, *AI vs. Machine Learning vs. Deep Learning vs. neural networks: What's the difference?*

[7] *Artificial neural network*. Jun. de 2023. URL: `https://en.wikipedia.org/wiki/Artificial_neural_network`.

realmente tomen la decisión correcta[8]. Esto se hace mediante unos algoritmos de los cuales hablaremos más adelante, que son los que permiten que la red neuronal "aprenda". De forma muy resumida, los algoritmos hacen lo siguiente:[9]

1. La red neuronal empieza con pesos aleatorios, y toma una decisión.
2. Se calcula el "coste" de la respuesta, que sería el equivalente a la cantidad de error de la respuesta.
3. El algoritmo cambia los valores de los pesos, para que este coste sea menor, es decir, la respuesta mejora.
4. Este proceso se repite hasta que la red neuronal tiene la precisión deseada.

Está claro que de la misma manera que hay incontables maneras de programar una inteligencia artificial, también hay una variedad de formas de programar una red neuronal:

Deep Learning

El término Deep Learning suele confundirse con las redes neuronales, a causa de su extrema similitud. El deep learning es un subconjunto de las redes neuronales, con la diferencia de que en el deep learning se suelen añadir múltiples capas ocultas a la red neuronal, pero excepto esta diferencia son esencialmente lo mismo.[10]

Convolutional Neural Networks

Una variación de las redes neuronales serían los *convolutional neural networks*, que son el modelo de red neuronal principalmente usado en la visión computacional. Estos destacan por ser capaces de reconocer patrones que se verían en una imagen, como por ejemplo, rectas, curvas, bordes, o incluso ojos o caras. Es con este modelo de red neuronal que las cámaras detectan las caras a la hora de hacer una foto.[11]

[8] Kavlakoglu, *AI vs. Machine Learning vs. Deep Learning vs. neural networks: What's the difference?*

[9] Grant Sanderson. *Neural Networks*. Oct. de 2017. URL: `https://youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi`.

[10] Kavlakoglu, *AI vs. Machine Learning vs. Deep Learning vs. neural networks: What's the difference?*

[11] Thomas Wood. *Convolutional Neural Network*. Mayo de 2019. URL: `https://deepai.org/machine-learning-glossary-and-terms/convolutional-neural-network`.

Recurrent Neural Networks

Otra variación de las redes neuronales serían las *recurrent neural networks* que estas destacan por usar datos secuenciados, que en esencia le aporta una "memoria" a la red neuronal. Los datos secuenciados permiten a la red neuronal tomar los inputs de decisiones anteriores de forma que puedan afectar los próximos outputs. Este modelo de red neuronal es comúnmente utilizado en asistentes virtuales como Siri o Alexa a la hora de procesar el lenguaje humano a algo que el ordenador entienda, o lo que se conoce como *natural language processing*.[12]

De todo lo previamente mencionado, esto es solo una pequeña parte de lo que se considera una inteligencia artificial, y queda fuera del alcance de este trabajo explicar de forma detallada como funciona cada una de ellas.

El objetivo de este trabajo, es dar un primer paso para entender un poco mejor cómo funcionan estas inteligencias artificiales. Es por esta razón, que durante este trabajo, nos centraremos específicamente en las redes neuronales. Como hemos visto, estas pueden llegar a ser extremadamente versátiles, dado que son usadas para la visión computacional, el *natural language processing*, o incluso gran parte del *machine learning*. Esta versatilidad sitúa a las redes neuronales como un magnífico primer paso para entender mejor como funcionan estas tecnologías.

2.3. Estructura de una red neuronal

Es importante destacar, que las redes neuronales también son conocidas como redes neuronales artificiales, para distinguirlas de la red neuronal que es nuestro cerebro. Esta similitud no es una mera causalidad, dado que gran parte de la motivación para crear una inteligencia artificial, fue descubrir que el cerebro era una red de neuronas, que funcionaba con pulsos eléctricos, y naturalmente surge la meta de simular esta red de neuronas artificialmente.[13]

Con los conocimientos actuales de neurología, sabemos que la estructura del cerebro es mucho más compleja que una red neuronal artificial, y a lo largo de los años la meta ya no ha sido simular el cerebro, sino obtener resultados más precisos. Sin embargo, esta relación nos permite empezar a entender una red neuronal.[14]

En el cerebro, las neuronas están conectadas entre sí, y transmiten pulsos eléctricos. En el caso de una red neuronal es lo mismo, los nodos están conec-

[12] *What are recurrent neural networks?* URL: `https://www.ibm.com/topics/recurrent-neural-networks`.

[13] *History of artificial intelligence.*

[14] *Artificial neural network.*

tados entre sí, y transmiten información. Es por esto que los nodos también son conocidos como neuronas o neuronas artificiales. ¿Pero qué es realmente un nodo?

2.3.1. Nodos

Un nodo, o neurona, es una variable con un valor entre 1 y 0, y al menos en este trabajo, es la manera en la que enviaremos y recibiremos información en la red neuronal. **De tal manera que $x \in [0, 1]$, donde x son los valores que puede tener el nodo. El valor que sostienen los nodos es conocido como activación, dado que se suele decir que el nodo se activa.**[15] Estos nodos están estructurados en capas, clasificadas en tres tipos. Tenemos la capa de input, que es la manera en la que transmitimos información a la red neuronal. La capa de input está conectada a capas ocultas, que modifican los valores de los nodos, para que cuando lleguen a la capa de output, nos den nuestros valores deseados. Esto forma la estructura de la figura 3, donde encontramos una red neuronal con una capa input, dos capas ocultas y una capa de output.

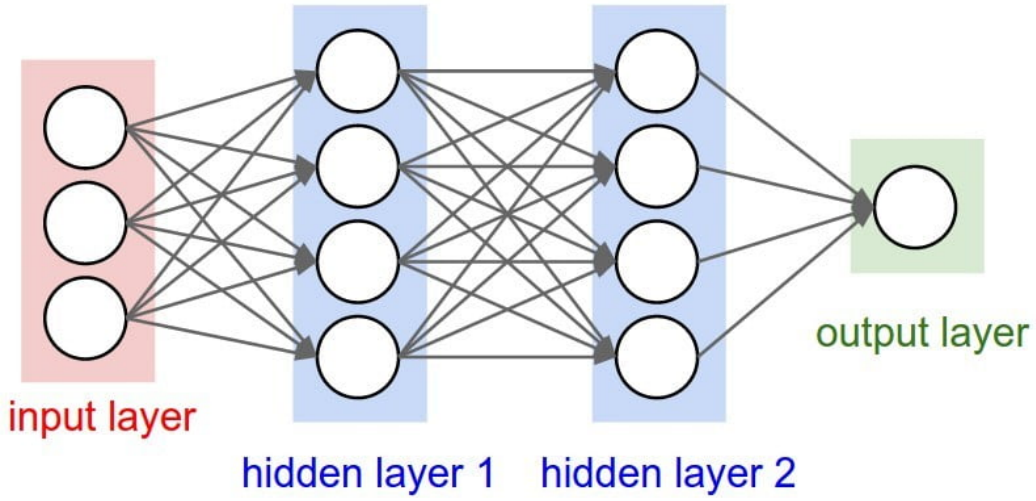


Figura 3: Ejemplo de la estructura de una red neuronal con dos capas ocultas

Es muy importante tener en cuenta que los nodos solo guardan valores entre 0 y 1, y al menos en las redes neuronales que veremos en este trabajo, necesitamos transmitir información con estas condiciones, con valores entre 1 y 0.

[15] Sanderson, *Neural Networks*.

Pongamos que la figura anterior nos predice si mañana lloverá o no. Los nodos de input sostienen valores entre 0 y 1 que pueden representar si hoy está lloviendo, si hace sol, o si hace mucho viento, donde 0 es que no ocurre, y 1 es que si, para este ejemplo. Estos valores pasan por las capas ocultas que modifican estos valores, para que la activación del nodo final sea un 1 si va a llover, un 0 si no va a llover, o cualquier valor entre medio, como 0,54, si no está seguro del todo.

2.3.2. Interacciones entre capas

La esencia de la red neuronal es cómo altera los valores del input para que el output sea el deseado. Debemos entender que los nodos tienen dos formas de alterar los valores.

Pero primero debemos entender lo fundamental de como se calcula los valores de los nodos. Observemos el siguiente ejemplo de red neuronal, sin capas ocultas:

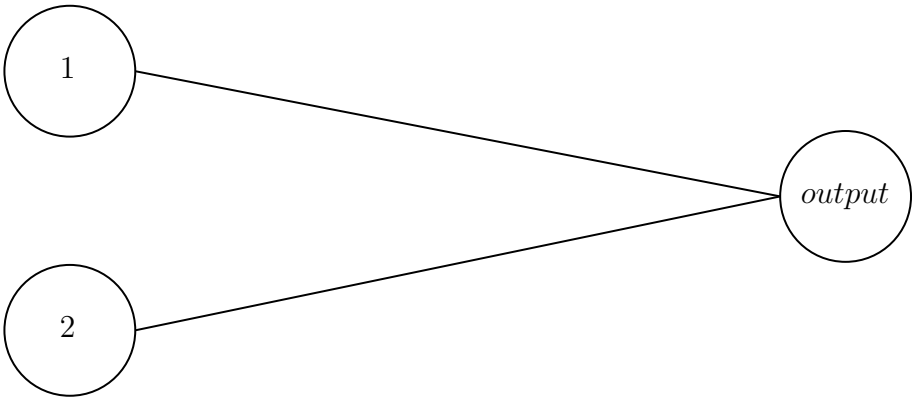


Figura 4: Estructura de red neuronal sin capa oculta

En este caso, el output se calcularía de la siguiente manera:

$$output = input1 + input2$$

De esta operación, saltan a la vista dos cosas: en primer lugar, con esta operación podría darnos valores mayores que 1, y por otro lado, los valores no se modifican, sino que solo se suman. El primero de estos problemas lo solucionamos con una función de activación ($\sigma(x)$):

$$output = \sigma(input1 + input2)$$

Una función de activación, convierte números reales a números entre 0 y 1. En este caso usamos la función sigmoide, representada como $\sigma(x)$, que para cualquier x, obtenemos una y, tal que $y \in (0, 1)$, como se ve en la figura 5. Existen otras funciones de activación.[16]

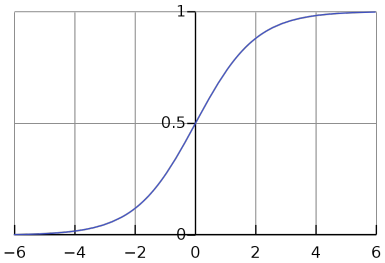


Figura 5: Función sigmoide

Pero seguimos sin modificar de forma sustancial los valores de los nodos. **Esto se hace gracias a los pesos (del inglés *weights*) que existen entre las conexiones**. Si volvemos al modelo representado anteriormente, si le añadimos pesos, obtendremos los valores con la siguiente operación:

$$output = \sigma(input1 * peso1 + input2 * peso2)$$

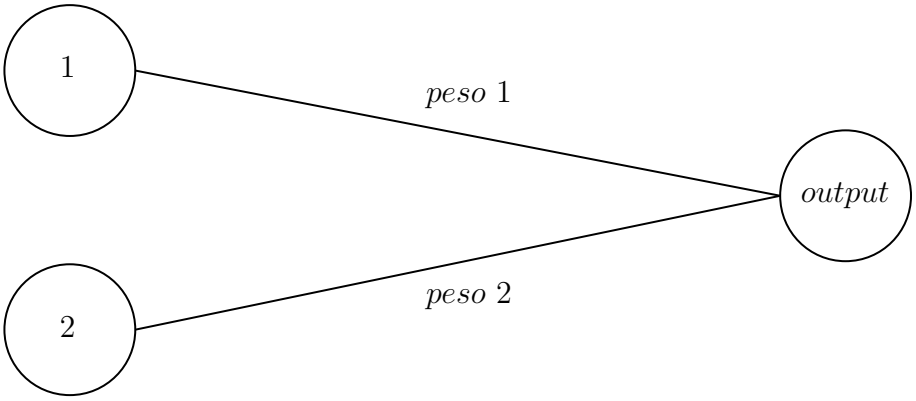


Figura 6: Estructura de red neuronal con pesos

De esta forma obtenemos una suma ponderada de los diferentes nodos, priorizando ciertas conexiones. Sin embargo, los valores de los nodos de input se modifican muy poco, de forma que en un caso perfecto donde tenemos los pesos bien definidos, tampoco tendríamos un modelo muy preciso. Vamos a añadir más capas al modelo para ver cómo cambia:

[16] *Activation function*. Mayo de 2023. URL: `https://en.wikipedia.org/wiki/Activation_function`.

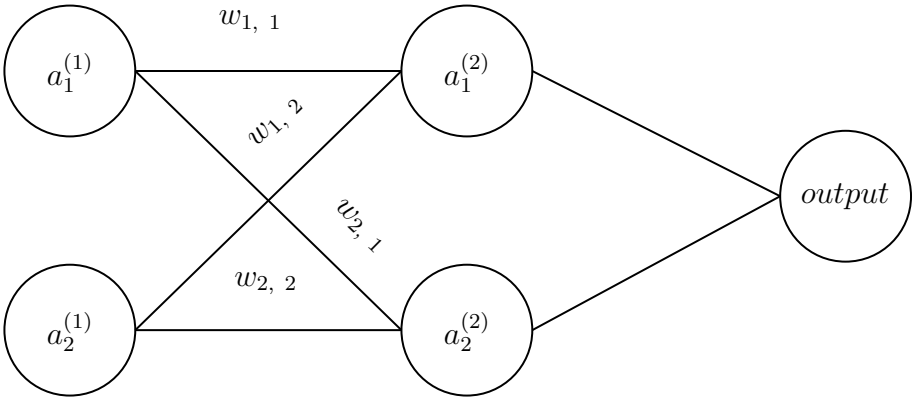


Figura 7: Red neuronal con notación correcta y capa oculta

Para no confundirnos con tantos nodos, vamos a cambiar nuestra forma de representarlos:

- La activación de cada nodo, será a_n^m, donde n es la posición del nodo en su capa, y m la capa del nodo.
- El peso de las conexiones, será $w_{p,n}$, donde p es la posición del nodo de la siguiente capa, y n la posición del nodo de la capa anterior.

Sabiendo esto, podemos calcular la activación del primer nodo de la capa oculta, $a_1^{(2)}$, de la siguiente manera:

$$a_1^{(2)} = \sigma(a_1^{(1)} w_{1,1} + a_2^{(1)} w_{1,2})$$

Esta ecuación nos permite calcular la activación de $a_1^{(2)}$, pero esto se tiene que repetir para $a_2^{(2)}$, y para el output. Sin embargo, aún podemos modificar las activaciones de una forma más para obtener resultados más precisos.

Pongamos por ejemplo un modelo más complejo, que nos predice si va a nevar, con una estructura similar al de la figura 3:

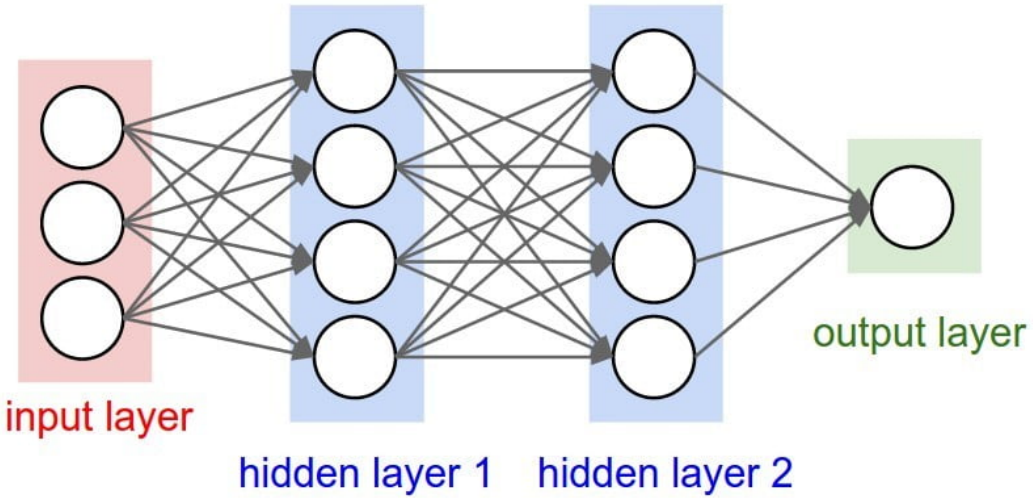


Figura 3: Ejemplo de estructura de red neuronal (repetida de la página 14)

Los pesos en una red neuronal de verdad estarían determinados por un algoritmo que veremos más adelante. Supongamos que se han determinado manualmente para que la primera capa oculta determine si las condiciones para que nieve se cumplan, y la segunda, si se van a mantener.

Esta idea de que cada capa determina una cosa distinta no se aleja mucho de lo que realmente pasa en una red neuronal. Sin embargo, como los pesos se ven determinados por un algoritmo, un humano nunca sería capaz de determinar los mismos pesos que el algoritmo, dado que no son nada obvios, y porque pueden haber miles de pesos.[17]

Volviendo a nuestro modelo que predice si va a nevar, sería importante resaltar ciertas conexiones entre nodos. Por ejemplo, si un nodo representa que hace viento y otro nodo que hace frío, estaría bien dar más prioridad a esa conexión de nodos específica. Esto se consigue gracias a un sesgo (del inglés *bias*), que nos permite priorizar ciertas conexiones frente a otras, para obtener mejores resultados.

De forma que para calcular $a_1^{(2)}$, nos quedaría una ecuación como la siguiente, donde $b_1^{(2)}$ es el sesgo del nodo $a_1^{(2)}$.[18]

$$a_1^{(2)} = \sigma(a_1^{(1)} w_{1,1} + a_2^{(1)} w_{1,2} + b_1^{(2)})$$

Como hemos visto previamente, esta ecuación solo nos calcula la activación de un nodo. En nuestro caso inicial, donde solo tenemos una capa oculta

[17] Sanderson, *Neural Networks*.

[18] Kavlakoglu, *AI vs. Machine Learning vs. Deep Learning vs. neural networks: What's the difference?*

con dos nodos, podemos manejarnos con la ecuación previamente mencionada. Sin embargo, para redes con miles de nodos, no es práctico ir repitiendo la misma ecuación miles de veces, y dependemos de las matrices para facilitarnos el cálculo.

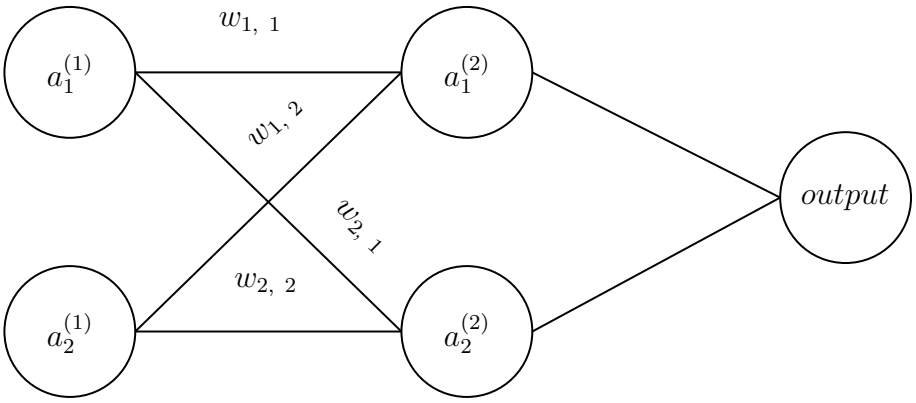


Figura 7: Red neuronal con notación correcta (repetida de la página 17)

Con el modelo de la figura 7, podemos organizar los pesos en una matriz, donde cada fila, son los pesos que afectan a un nodo en la siguiente capa. En nuestro modelo, en la primera fila de la matriz tendríamos los pesos que afectan a $a_1^{(2)}$ y en la segunda fila los que afectan a $a_2^{(2)}$:

$$W = \begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{bmatrix}$$

Esta matriz, la podemos multiplicar por un vector con todos los nodos de la capa anterior, y podemos sumarle otro vector con los sesgos de la capa, de la siguiente manera:

$$\begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{bmatrix} * \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \end{bmatrix} + \begin{bmatrix} b_1^{(2)} \\ b_2^{(2)} \end{bmatrix}$$

Esta expresión se puede colocar dentro de la función sigmoide y obtendremos un vector con las activaciones de los nodos de la siguiente capa:

$$\begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \end{bmatrix} = \sigma\left(\begin{bmatrix} w_{1,1} & w_{1,2} \\ w_{2,1} & w_{2,2} \end{bmatrix} * \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \end{bmatrix} + \begin{bmatrix} b_1^{(2)} \\ b_2^{(2)} \end{bmatrix}\right)$$

Estas matrices y vectores, se pueden generalizar para cualquier número de nodos, pesos o sesgos:

$$\begin{bmatrix} a_1^{(2)} \\ a_2^{(2)} \\ \vdots \\ a_n^{(2)} \end{bmatrix} = \sigma(\begin{bmatrix} w_{1,1} & w_{1,2} & \dots & w_{1,n} \\ w_{2,1} & w_{2,2} & \dots & w_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{p,1} & w_{p,2} & \dots & w_{p,n} \end{bmatrix} * \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \\ \vdots \\ a_n^{(1)} \end{bmatrix} + \begin{bmatrix} b_1^{(2)} \\ b_2^{(2)} \\ \vdots \\ b_n^{(2)} \end{bmatrix})$$

Esta ecuación se puede simplificar con un cambio de variable. Podemos llamar W a la matriz de los pesos, $a^{(n)}$ al vector con las activaciones de la capa n, y $b^{(n)}$ al vector de los sesgos de la capa n, para obtener la siguiente ecuación:[19]

$$a^{(2)} = \sigma(W * a^{(1)} + b^{(2)})$$

De este modo, se pueden condensar todos los pesos y sesgos de una capa en esta ecuación.

2.3.3. Ejemplo de la estructura

Para entender estas conexiones, y ver cómo pueden afectar al resultado, vamos a poner un ejemplo de red neuronal, en el que determinaremos manualmente los pesos y los sesgos.

Imaginemos que nos queremos comprar una cámara para hacer fotos, pero no sabemos cuál. De esta forma, ideamos una red neuronal para que decida si es una buena compra para nosotros.

En primer lugar, vamos a definir cómo queremos que sea nuestra cámara:

1. Como queremos la cámara para usarla de vez en cuando, no valoramos mucho que haga fotos con la máxima calidad posible.

2. Pero sí que sería importante que la batería fuese buena, para no tener que preocuparnos de cargarla.

Sabiendo esto podríamos crear un modelo como se ve en la figura 8. Esta red neuronal, no sería muy útil en la realidad, porque no tiene suficiente información, sin embargo, nos permite entender mejor el comportamiento de una red neuronal:

[19] Sanderson, *Neural Networks*.

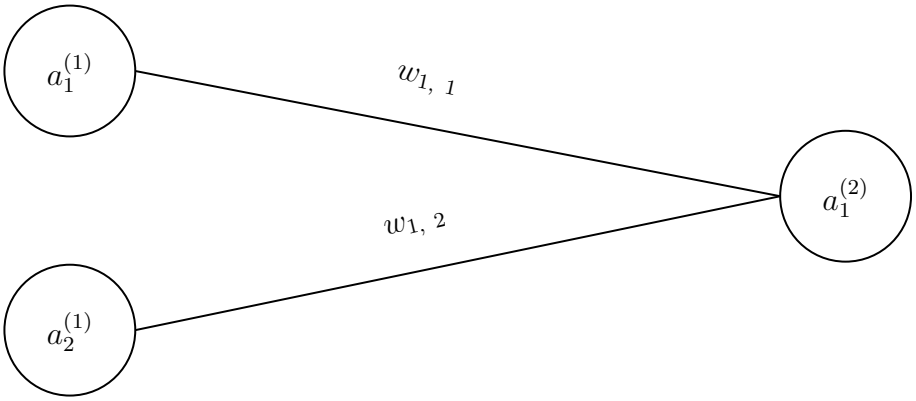


Figura 8: Modelo de ejemplo para decidir entre modelos de cámaras

Para este ejemplo, podemos establecer lo siguiente:

- $a_1^{(1)}$ representa la calidad de las fotos, siendo 1 la mejor calidad del mercado, y 0 la peor.
- $a_2^{(1)}$ representa la batería de la cámara, siendo 1 la mejor batería del mercado, y 0 la peor.

Los pesos y sesgos de la red neuronal, normalmente son determinados por un algoritmo que veremos más adelante. A diferencia de los nodos, los valores de los pesos y sesgos, no están limitados a un rango muy estricto, porque el algoritmo poco a poco modificará los pesos hasta su valor deseado.

Generalmente, para no dar mucho trabajo a la red neuronal, se suele empezar con pesos entre $-0,5$ y $0,5$, para que la red neuronal decida para donde quiere modificar los pesos más allá de esos valores.[20] [21]

Pero como aún no hemos visto como funciona el algoritmo que los determina, vamos a determinarlos manualmente con el siguiente criterio:

- Como no nos importa mucho la calidad, $w_{1,1} = 2$
- Como queremos una batería decente, $w_{1,2} = 10$
- Establecemos un sesgo de -10 en el output para que la cámara tenga un mínimo de calidad

[20] Bot Academy. *Neural Networks explained from scratch using Python*. Ene. de 2021. URL: https://www.youtube.com/watch?v=9RN2Wr8xvro&t=853s.

[21] Sebastian Lague. *How to create a neural network (and train it to identify doodles)*. Ago. de 2022. URL: https://www.youtube.com/watch?v=hfMk-kjRv4c.

De esta forma, damos más importancia a la batería, es decir, si encontramos una cámara con muy buena calidad, pero mala batería, la red neuronal no la recomendará, pero una con una batería buena y mala calidad, sí que la recomendará.

Un sesgo de -10 puede parecer extraño, pero nos permite que haya un cierto balance entre los dos criterios, la calidad y la batería. A continuación, veremos 3 ejemplos de camaras, y veremos qué recomendaciones nos devuelve la red neuronal para cada una de ellas. Más adelante, en el ejemplo de la cámara 2, como el sesgo participa en el resultado.

Hemos encontrado los siguientes modelos, y vemos si la red neuronal los recomienda:

Cámara 1. Hemos encontrado una cámara con la mejor calidad, y una batería decente. Por lo tanto, a $a_1^{(1)}$ le daríamos un valor de 1, dado que representa la calidad, y a $a_2^{(1)}$ un 0.6, dado que representa la batería de la cámara. De esta forma, el vector de activaciones sería el siguiente:

$$a^{(1)} = \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \end{bmatrix} = \begin{bmatrix} 1 \\ 0{,}6 \end{bmatrix}$$

Los pesos los hemos establecido previamente como los siguientes:

$$W = \begin{bmatrix} w_{1,1} \\ w_{1,2} \end{bmatrix} = \begin{bmatrix} 2 \\ 10 \end{bmatrix}$$

Como solo tenemos un nodo de output, no hace falta hacer operaciones con matrices, y podemos usar la siguiente ecuación:

$$a_1^{(2)} = \sigma(a_1^{(1)} w_{1,1} + a_2^{(1)} w_{1,2} + b_1^{(2)})$$

$$a_1^{(2)} = \sigma(1 * 2 + 0{,}6 * 10 - 10)$$

$$a_1^{(2)} = \sigma(-2)$$

$$a_1^{(2)} \approx 0{,}119$$

Como nos da un valor muy bajo, podemos interpretarlo como que la red neuronal no recomienda esta cámara. Como los pesos que hemos establecido valoran más la batería que la calidad, no nos la recomienda.

Cámara 2. Hemos encontrado otra cámara con una de las mejores baterías del mercado, pero con una calidad pésima. Por lo tanto, a $a_1^{(1)}$ le daríamos un valor de 0.1, dado que representa la calidad, y a $a_2^{(1)}$ un

0.9, dado que representa la batería de la cámara. Obtenemos el siguiente vector con las activaciones:

$$a^{(1)} = \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \end{bmatrix} = \begin{bmatrix} 0{,}1 \\ 0{,}9 \end{bmatrix}$$

Y los mismos pesos que antes:

$$W = \begin{bmatrix} w_{1,1} \\ w_{1,2} \end{bmatrix} = \begin{bmatrix} 2 \\ 10 \end{bmatrix}$$

Con la misma ecuación de antes, obtenemos el siguiente resultado:

$$a_1^{(2)} = \sigma(a_1^{(1)} w_{1,1} + a_2^{(1)} w_{1,2} + b_1^{(2)})$$

$$a_1^{(2)} = \sigma(0{,}1 * 2 + 0{,}9 * 10 - 10)$$

$$a_1^{(2)} = \sigma(-0{,}8)$$

$$a_1^{(2)} \approx 0{,}310$$

Vemos que esta cámara la recomienda más, sin embargo, sigue sin ser una recomendación absoluta. Vemos que el sesgo que hemos incorporado implica que la cámara tiene que tener un mínimo de calidad para que sea recomendada, y es precisamente esto lo que nos ha pasado.

Gracias al sesgo, una cámara no puede ser recomendada únicamente por su batería, que es el criterio que hemos priorizado con los pesos, sino que también necesita llegar a un mínimo de calidad.

Cámara 3. Hemos encontrado una tercera cámara, esta vez, con una batería similar a la anterior, pero con una calidad aceptable. Por lo tanto, a $a_1^{(1)}$ le daríamos un valor de 0.6, dado que representa la calidad, y a $a_2^{(1)}$ un 0.9, dado que representa la batería de la cámara. Obtenemos el siguiente vector con las activaciones:

$$a^{(1)} = \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \end{bmatrix} = \begin{bmatrix} 0{,}6 \\ 0{,}9 \end{bmatrix}$$

Y los mismos pesos que antes:

$$W = \begin{bmatrix} w_{1,1} \\ w_{1,2} \end{bmatrix} = \begin{bmatrix} 2 \\ 10 \end{bmatrix}$$

Con la misma ecuación de antes, obtenemos el siguiente resultado:

$$a_1^{(2)} = \sigma(a_1^{(1)} w_{1,1} + a_2^{(1)} w_{1,2} + b_1^{(2)})$$

$$a_1^{(2)} = \sigma(0{,}6 * 2 + 0{,}9 * 10 - 10)$$
$$a_1^{(2)} = \sigma(0{,}2)$$
$$a_1^{(2)} \approx 0{,}550$$

Vemos de este resultado, que recomienda la tercera cámara más que las otras. Es importante destacar que las activaciones totales de la primera cámara, eran mayores que las de la tercera cámara. Esto nos lleva a pensar que la primera cámara es mejor.

Cámara 1

$$a^{(1)} = \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \end{bmatrix} = \begin{bmatrix} 1 \\ 0{,}6 \end{bmatrix}$$

Cámara 3

$$a^{(1)} = \begin{bmatrix} a_1^{(1)} \\ a_2^{(1)} \end{bmatrix} = \begin{bmatrix} 0{,}6 \\ 0{,}9 \end{bmatrix}$$

$$1 + 0,6 = 1,6 > 1,5 = 0{,}9 + 0,6$$

Pero es gracias a los pesos, que hemos determinado para que prioricen la batería por encima de la calidad, que la recomendación de la cámara 3 es considerablemente más alta que la de la cámara 1.

En este ejemplo, dado que solo se han tenido que determinar dos pesos y un sesgo, no resulta muy complicado idear unos valores razonables para que nuestra red neuronal cumpla con su función. Pero para redes neuronales con miles de nodos, la tarea de determinar manualmente los pesos y sesgos se convierte en una tarea muy compleja. Otro problema que surge es la manera en la que se determinan estos valores. Para nuestro ejemplo, como sabemos previamente que valoramos la batería más que la calidad, resulta obvio dar más peso a la batería. ¿Pero en función de qué determinamos los pesos y sesgos para identificar números o letras escritos a mano? ¿Cómo sabemos si los pesos que hemos determinado mejoran o empeoran los resultados? ¿Cómo hacemos que la red neuronal aprenda?

2.4. ¿Cómo aprende una red neuronal?

Veremos a lo largo de este trabajo que ,**cuando hablamos de que una red neuronal aprende, solo estamos buscando el mínimo de una función.** La dificultad aparece en encontrar la funcion que queremos minimizar, y en que estas funciones pueden incluir miles de variables, que debemos tener en cuenta.

Pero explicaremos más adelante cómo llegamos a esta función, y cómo se relaciona el mínimo de esta función con el nivel de aprendizaje. De momento, nos basta con saber que para minimizar esta función, como depende de miles de variables, encontrar el mínimo no es tan facil como igualar la derivada a 0. Usaremos un proceso conocido como descenso de gradiente (del inglés *gradient descent*) donde dependemos de muchos datos para encontrar el mínimo, y que la red neuronal aprenda.[22]

Si queremos que una red neuronal aprenda a leer dígitos o letras escritas a mano, por ejemplo, necesitamos **miles de ejemplos que estén etiquetados** de forma que una imagen de un 2 esté etiquetada con un dos.

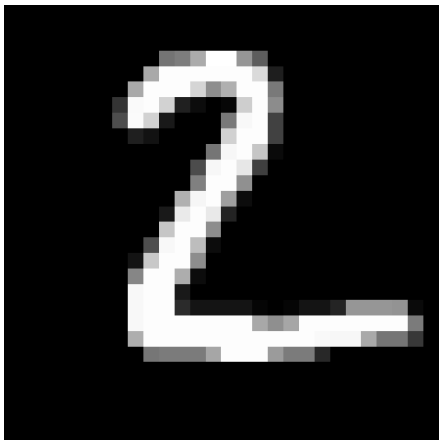


Figura 9: Imagen de un 2 etiquetada con un 2

Los datos se dividen en dos grupos: el de entrenamiento y el de prueba. Los datos de entrenamiento, son los datos de los que la red neuronal aprenderá, y son los que usaremos para encontrar el mínimo. Los de prueba son los que validarán los resultados de la red neuronal, con datos que la red neuronal no ha visto antes. Los datos de prueba nos muestran cómo respondería la red neuronal ante datos de los que no sabemos la respuesta.[23]

Por ejemplo, si una red neuronal predice si una persona tiene una enfermedad o no, la entrenaríamos con unos datos de entrenamiento de pacientes anteriores, donde sabemos si han sido diagnosticados con la enfermedad. Esto nos permite ver si la red neuronal acierta o se equivoca, y entrenarla.

Una vez la hemos entrenado, pasamos datos de prueba que no ha visto antes la red neuronal para validar los resultados. Estos datos también son

[22] *Gradient*. Jun. de 2023. URL: `https://en.wikipedia.org/wiki/Gradient`.

[23] *Training, validation, and test data sets*. Jul. de 2023. URL: `https://en.wikipedia.org/wiki/Training,_validation,_and_test_data_sets`.

de pacientes anteriores, para ver si la red neuronal acierta o no. En la validación, no buscamos cambiar los pesos y sesgos de la red neuronal porque en teoría, una vez ha sido entrenada, debería de ser capaz de diagnosticar la enfermedad.

Si tras validar la red neuronal vemos que tiene una precisión del 95 % , podemos asumir que la red neuronal identificará la enfermedad correctamente el 95 % de las veces. Ahora, si le damos datos de un paciente nuevo, del cual no sabemos si tiene la enfermedad, la red neuronal es capaz de identificar los mismos patrones que ha identificado en los datos de entrenamiento, y diagnosticaría al paciente con una precisión del 95 %.

En esta explicación, sin embargo, hemos simplificado muchos de los conceptos necesarios para entrenar una red neuronal, que son los que veremos más adelante.

2.4.1. Coste

Como acabamos de explicar, para entrenar a una red neuronal, debemos decirle a la red neuronal si la respuesta que nos ha dado es la correcta o la incorrecta. **El coste es precisamente esto, nos cuantifica cómo de incorrecta es la respuesta de la red neuronal**. Como los nodos tienen activaciones continuas, es decir, que son valores entre 0 y 1, no podemos determinar directamente si la red neuronal acierta o falla, así que medimos el grado de fallo.

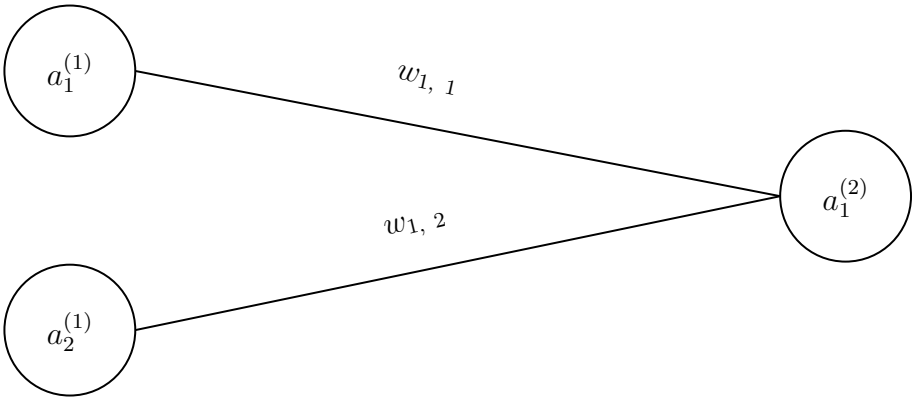


Figura 10: Modelo de ejemplo para identificar enfermedades

Volvamos a la red neuronal que diagnostica enfermedades. Pongamos el caso donde le hemos introducido los datos de un paciente que sí que ha sido diagnosticado con la enfermedad, pero nuestro modelo, todavía sin entrenar, da una activación de 0,33 en el output. Calculamos el coste de un solo nodo

de la siguiente manera, donde $a_2^{(1)}$ es la predicción de la red neuronal, 0, 33 en este caso, y $ValorDeseado(a_2^{(1)})$, es la respuesta correcta, en este caso 1 porque el paciente sí ha sido diagnosticado:[24] [25]

$$Cost(a_2^{(1)}) = (a_2^{(1)} - ValorDeseado(a_2^{(1)}))^2$$
$$Cost(a_2^{(1)}) = (0{,}33 - 1)^2$$
$$Cost(a_2^{(1)}) = 0{,}4489$$

Hacemos el cuadrado de la diferencia entre la activación obtenida, y el valor deseado. Como sabemos que el paciente ha sido diagnosticado con la enfermedad, la activación esperada era de 1. Pero como la red neuronal solo ha devuelto una activación de 0.33, el coste es relativamente alto, por lo tanto, habría que entrenarlo más.

Esto solo es para un nodo, pero también es común tener más de un nodo en el output. Para calcular el coste total de la red neuronal, se suma el coste de los nodos individualmente:

$$Cost = \sum_{i=1}^{n} (a_i^{(output)} - ValorDeseado(a_i^{(output)}))^2$$

Se puede observar que el coste mide el grado de error que produce nuestra red neuronal, de forma que cuanto más bajo sea el error, más precisa es nuestra red neuronal. Si volvemos al ejemplo de antes, vemos que para un mismo caso, lo único que modifica el coste son los pesos y los sesgos. Es decir que existe una combinación de pesos y sesgos ideales, donde el coste es el mínimo. Esto se da porque de la forma que hemos calculado el coste, este depende de las activaciones de los nodos, que dependen de los pesos y los sesgos. Por lo tanto, podemos obtener el coste en función de los pesos y sesgos de nuestra red neuronal.

$$Cost(w_{1,1}, w_{1,2}, b_1^{(2)}) = 0{,}4489$$

Si podemos encontrar los valores que necesitan los pesos y sesgos para que el coste sea el mínimo, tendríamos nuestra red neuronal entrenada. Pero a la que se le añaden pesos y sesgos, esto se complica rápidamente.

La dependencia del coste en función de los pesos y sesgos puede ser representado gráficamente. En este caso previo que hemos analizado, sería imposible porque obtendríamos una gráfica en 4 dimensiones, pero si solo tuviésemos 1 o 2 pesos, sí que sería posible. Esto, en práctica, no nos sirve de mucho

[24] Sanderson, *Neural Networks*.
[25] Lague, *How to create a neural network (and train it to identify doodles)*.

porque las redes neuronales pueden tener miles de pesos y sesgos. Aun así, el poder representar el coste en una gráfica nos ayuda a entender mejor el siguiente concepto, el descenso de gradiente.

2.4.2. Descenso de Gradiente del coste

En el siguiente apartado, veremos el concepto de descenso de gradiente. Este método nos sirve para encontrar los mínimos en funciones complejas, con miles de variables, que no podemos representar en gráficas de 2 o 3 dimensiones.

Sin embargo, empezaremos explicando el descenso de gradiente con funciones de 2 dimensiones, y después expandiremos a funciones de 3 dimensiones para entender cómo se expandiría a más dimensiones.

2.4.2.1 Descenso de gradiente aplicado a 2 dimensiones:

Como se ha comentado anteriormente, podemos representar el coste en una gráfica. Esto es muy útil para minimizar el peso, y mejorar la precisión de nuestra red neuronal. Con una gráfica podemos visualizar cómo reduce el coste la red neuronal, aunque nos sirva solo para redes muy simples. Pongamos una red neuronal con un solo peso, de forma que el coste dependa de un solo peso.

Podemos hacernos una idea de como sería la gráfica operando. Sabemos que la activación del nodo del output, $a_1^{(2)}$ por ejemplo, se calcula de la siguiente manera, de momento sin sesgos ni funciones de activación.

$$a_1^{(2)} = a_1^{(1)} w_{1,1}$$

Y sabemos que el coste se calcula con $a_1^{(2)}$:

$$Cost = (a_1^{(2)} - ValorDeseado(a_1^{(2)}))^2$$

Podemos juntar estas dos ecuaciones, pero por ahora quitaremos los sesgos y la función sigmoide para simplificar las cosas:

$$Cost = (a_1^{(1)} w_{1,1} - ValorDeseado(a_1^{(2)}))^2$$

Como solo queremos hacernos una idea de la gráfica que estamos buscando, podemos dar valores a $a_1^{(1)}$ y $ValorDeseado(a_1^{(2)})$, por ejemplo, de 0.3 y 1 respectivamente. Esto lo hacemos porque solo queremos ver las relaciones entre los pesos y el coste.

$$Cost = (0{,}3 w_{1,1} - 1)^2$$

$$Cost = 0{,}09(w_{1,1})^2 + 1 - 2 * 0{,}3w_{1,1}$$

$$Cost = 0{,}09(w_{1,1})^2 - 0{,}6w_{1,1} + 1$$

Vemos que nos queda una ecuación cuadrática en función de $w_{1,1}$:

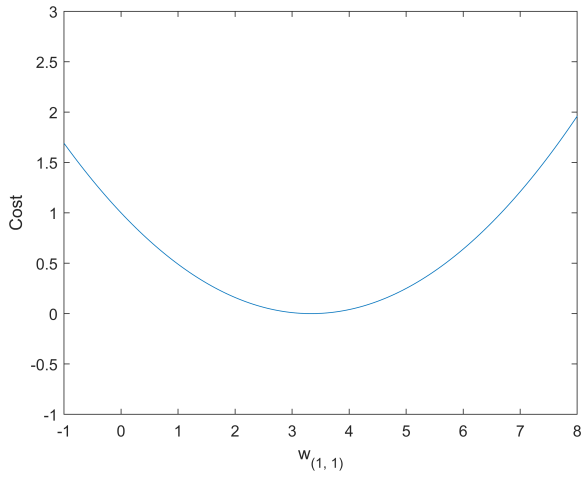


Figura 11: Función que expresa el coste en función de $w_{1,1}$

Pero lo que realmente buscamos es el peso que nos dé el coste más bajo. En este caso podemos igualar a 0 la derivada de la función que hemos encontrado y vemos que el peso ideal es de $3.\overline{3}$. Pero en una red neuronal de verdad con miles de pesos y sesgos, resolver una ecuación con miles de incógnitas no es ideal. Dependemos de otro método, que sí se puede aplicar a redes más complejas, el descenso de gradiente

Empezamos escogiendo un punto aleatorio de la gráfica, y medimos la pendiente. Si la pendiente es positiva, tenemos que bajar nuestro peso, si la pendiente es negativa, debemos aumentarlo. Si repetimos este proceso, acabaremos llegando a un mínimo.

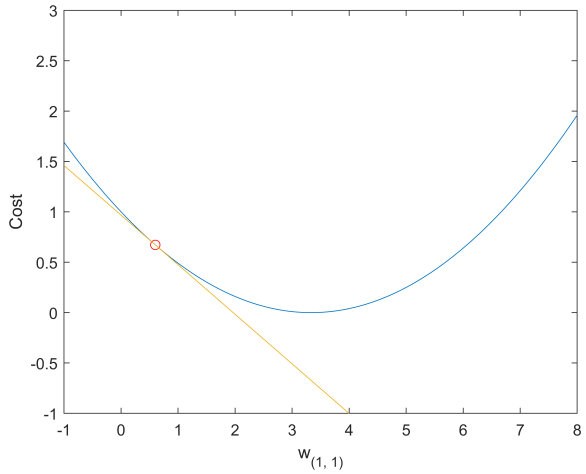


Figura 12: Función a la que aplicamos el descenso de gradiente

Hemos escogido un punto aleatorio, y vemos que la pendiente es negativa. Esto quiere decir que tenemos que aumentar el peso para llegar a un mínimo, es decir que llegamos a un punto donde la pendiente es 0.

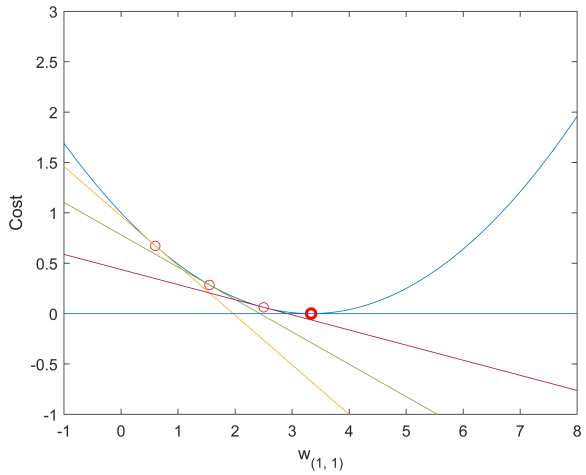


Figura 13: Función de la cual encontramos el mínimo

Vemos que siguiendo este método acabamos llegando a un mínimo en la gráfica. Esto se conoce como descenso de gradiente (del inglés *gradient descent*).

Este método, lo podemos aplicar a dimensiones más altas, es decir que lo podemos usar para nuestra red neuronal, pero tiene sus desventajas. La más obvia es el hecho que obtenemos un mínimo, pero este mínimo puede

no ser un mínimo global. En la figura 14, por ejemplo, siguiendo el descenso de gradiente, dependiendo de que punto escojamos, podemos acabar en un mínimo diferente al mínimo global.

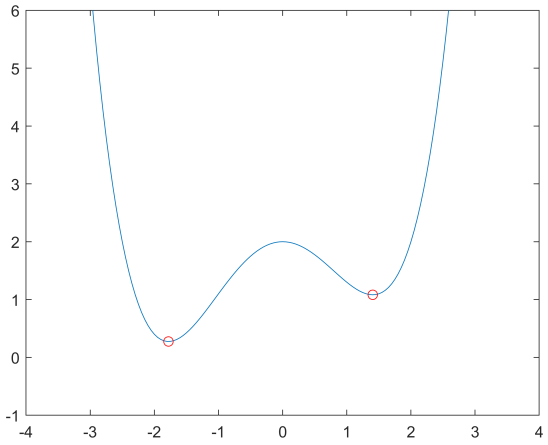


Figura 14: Función con múltiples mínimos

En la práctica, no supone una pérdida muy grande el no poder llegar al mínimo absoluto. Por otro lado, obtener el mínimo absoluto ya es complicado de por sí, sobre todo teniendo en cuenta que en la práctica se trabaja con miles de pesos y sesgos.

2.4.2.2 Descenso de gradiente aplicado a 3 dimensiones:

Vamos a aumentar un poco más la complejidad de nuestra red neuronal para ver cómo se aplica el descenso de gradiente a más dimensiones. De momento seguimos sin sesgos ni funciones de activación, pero añadimos un nodo, y su correspondiente peso:

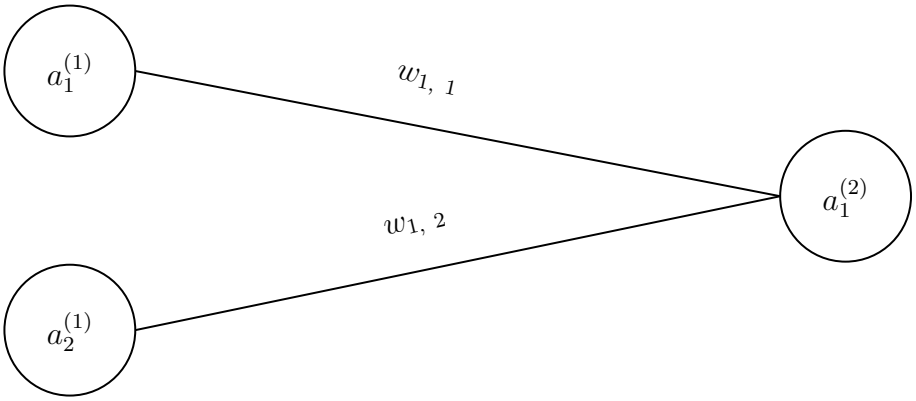


Figura 10: Red neuronal con dos nodos de entrada (repetida de la página 26)

Calcularíamos el coste de $a_1^{(2)}$ de la siguiente manera, de momento seguimos sin tener en cuenta los sesgos o la función sigmoide:

$$Cost = (a_1^{(2)} - ValorDeseado(a_1^{(2)}))^2$$
$$Cost = ((a_1^{(1)} w_{1,1} + a_2^{(1)} w_{1,2}) - ValorDeseado(a_1^{(2)}))^2$$

Como solo queremos ver las relaciones entre los pesos y el coste, podemos tratar a $a_1^{(1)}$, $a_2^{(1)}$ y $ValorDeseado(a_1^{(2)})$ como constantes, con valores de 0,3 0,4 y 1 por ejemplo:

$$Cost = (0{,}3w_{1,1} + 0{,}4w_{1,2} - 1)^2$$

Vemos que nos queda una ecuación de tres variables, que podemos representar en una gráfica:

En esta gráfica buscamos minimizar el coste, para obtener resultados más precisos. Si volvemos al ejemplo de dos dimensiones, buscábamos la pendiente, para saber como modificar los pesos para minimizar el coste. En las tres dimensiones, lo hacemos gracias al gradiente.

Con el gradiente, obtenemos un campo de vectores en el plano, que nos indica la dirección en la que nos tenemos que mover para que el coste aumente lo máximo posible. En nuestro caso, para reducirlo al mínimo, podemos movernos en la dirección opuesta a el gradiente. Vamos a calcularlo para hacernos una mejor idea:

$$Cost = (0{,}3x + 0{,}4y - 1)^2$$
$$Cost = 0{,}09x^2 + 0{,}24xy - 0{,}6x + 0{,}16y^2 - 0{,}8y + 1$$

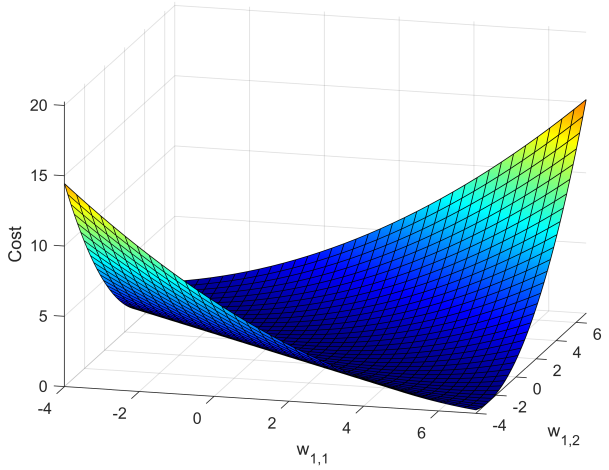


Figura 15: Función que expresa el coste en función de $w_{1,1}$ y $w_{1,2}$

Vamos a cambiar $w_{1,1}$ y $w_{1,2}$ por x y y, para facilitarnos el trabajo. El gradiente es un vector cuyos componentes son las derivadas parciales de f, en nuestro caso, las derivadas parciales del coste[26]:

$$Cost = 0{,}09x^2 + 0{,}24xy - 0{,}6x + 0{,}16y^2 - 0{,}8y + 1$$

$$\frac{\partial Cost}{\partial x} = 0{,}18x + 0{,}24y - 0{,}6$$

$$\frac{\partial Cost}{\partial y} = 0{,}24x + 0{,}32y - 0{,}8$$

Con las derivadas parciales del coste, podemos calcular el gradiente como un vector con las derivadas parciales como sus componentes. Simbolizamos el gradiente con ∇ para diferenciarlo de la función del coste.

$$\nabla Cost = \begin{bmatrix} \frac{\partial Cost}{\partial x} \\ \frac{\partial Cost}{\partial y} \end{bmatrix}$$

$$\nabla Cost = \begin{bmatrix} 0{,}18x + 0{,}24y - 0{,}6 \\ 0{,}24x + 0{,}32y - 0{,}8 \end{bmatrix}$$

Con este vector, para un punto (x, y), es decir, para una combinación de pesos, si sustituimos las variables de estas ecuaciones por los pesos, $w_{1,1}$ y $w_{1,2}$, obtenemos un vector que nos indica hacia qué dirección debemos modificar

[26] *Gradient*.

x o y, es decir, como tenemos que modificar los pesos, para maximizar el coste.[27] Como queremos minimizarlo, trabajaremos con $-\nabla Cost$.

Imaginemos que nuestra red neuronal tiene unos pesos de 5 para $w_{1,1}$, y de 4 para $w_{1,2}$. Con esto podemos calcular el coste, para ver como de precisa es nuestra red neuronal:

$$Cost = (0{,}3w_{1,1} + 0{,}4w_{1,2} - 1)^2$$
$$Cost = (0{,}3 * 5 + 0{,}4 * 4 - 1)^2$$
$$Cost = (1{,}5 + 1{,}6 - 1)^2$$
$$Cost = (2{,}1)^2$$
$$Cost = 4{,}41$$

Este punto lo podemos representar en nuestra gráfica, y vemos que se podría minimizar más el coste:

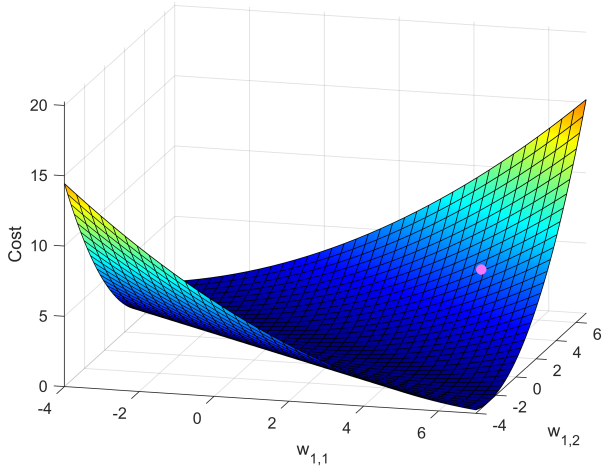


Figura 16: Función con el coste marcado cuando $w_{1,1} = 5$ y $w_{1,2} = 4$

Podemos también calcular el gradiente del coste, para saber como tenemos que mover los pesos para minimizar el coste, sabiendo que $x = w_{1,1}$, y $y = w_{1,2}$:

$$\nabla Cost = \begin{bmatrix} 0{,}18x + 0{,}24y - 0{,}6 \\ 0{,}24x + 0{,}32y - 0{,}8 \end{bmatrix}$$

[27] *Gradient.*

$$\nabla Cost = \begin{bmatrix} 0{,}18 * 5 + 0{,}24 * 4 - 0{,}6 \\ 0{,}24 * 5 + 0{,}32 * 4 - 0{,}8 \end{bmatrix}$$

$$\nabla Cost = \begin{bmatrix} 0{,}9 + 0{,}96 - 0{,}6 \\ 1{,}2 + 1{,}28 - 0{,}8 \end{bmatrix}$$

$$\nabla Cost = \begin{bmatrix} 1{,}26 \\ 1{,}68 \end{bmatrix}$$

$$-\nabla Cost = \begin{bmatrix} -1{,}26 \\ -1{,}68 \end{bmatrix}$$

Este vector nos indica hacia qué dirección debemos modificar los pesos para que el coste se acerque a 0. También vemos que este vector nos marca como de importante es un peso para bajar el coste. En este caso, observamos que cambios en el eje y, es decir, cambios en $w_{1,2}$, afectan más al coste que cambios en $w_{1,1}$. Podemos representar este vector en una gráfica y vemos que apunta hacia donde el coste es 0:

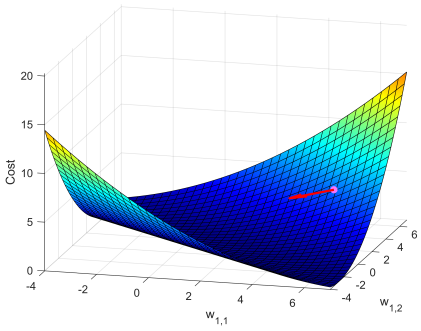


Figura 17: Perspectiva general.

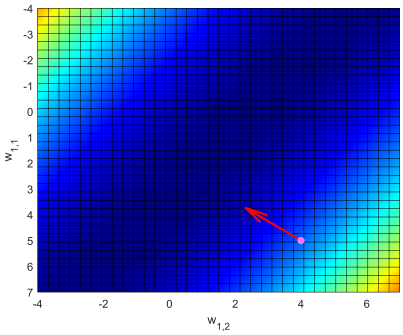


Figura 18: Perspectiva cenital.

En las figuras 17 y 18 , los colores oscuros representan los valores más bajos, y podemos ver cómo el gradiente del coste apunta hacia los valores más bajos. **Cabe destacar que la variación que impondremos a los pesos y los sesgos para acercarnos al mínimo será proporcional al gradiente que hemos calculado**. Multiplicaremos el gradiente por un número suficientemente pequeños para que el vector que obtengamos no se salte ningún mínimo. Esto lo hacemos porque, en ciertos casos, el vector que obtenemos del gradiente puede pasarse del mínimo. En nuestro caso, no tenemos que preocuparnos de esto, pero con miles de pesos y sesgos sí que puede llegar a ser un problema.

Cogeríamos los pesos, y le restaríamos un múltiplo del gradiente. Con esto obtenemos pesos nuevos que resultan en un coste menor, es decir, mayor precisión. **Esto es lo que se conoce como entrenar una red neuronal. Lo único que estamos haciendo es encontrar un mínimo**, aunque normalmente se trate de un mínimo en un espacio con miles de dimensiones.

Sin embargo, en esta situación solo lo hemos hecho con un caso de la red neuronal, cuando $a_1^{(1)}$ y $a_2^{(1)}$ son 0,3 y 0,4. **En teoría usaríamos una media de todos los costes con todos los datos a nuestra disposición. Cuando hacemos la media del descenso de gradiente, obtenemos pesos que en teoría reducen el coste para todos los datos que tenemos, y no solo para un caso en específico**. Si no hiciésemos una media de todos los costes, estaríamos dando a entender a la red neuronal que siempre devuelva la misma respuesta.

En práctica, sin embargo, no resulta eficiente analizar miles de datos para cada paso del descenso de gradiente, y veremos más adelante una variación del método de descenso de gradiente que mejora su eficiencia.

Pero lo que acabamos de ver, no lo podemos aplicar directamente a redes neuronales más complejas. Nuestro ejemplo, solo tenía un nodo de output, y ni siquiera tenía capas ocultas.

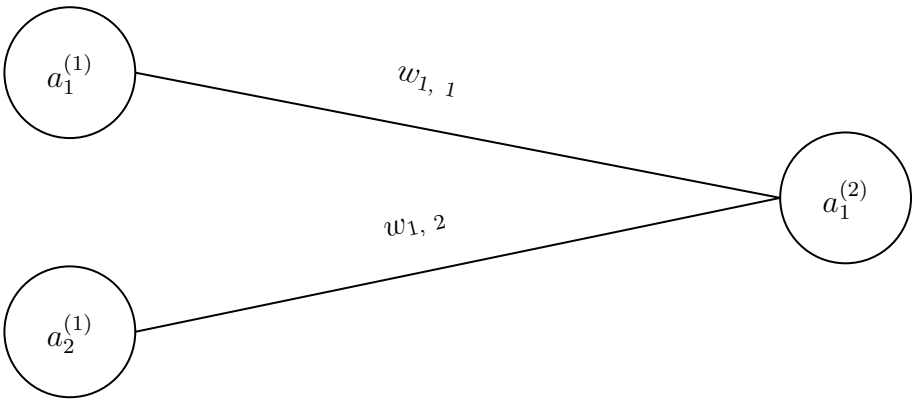


Figura 10: Red neuronal con dos nodos de entrada (repetida de la página 26)

Esto nos ha permitido calcular el gradiente del coste, sin mucha complicación. Pero la situación es diferente cuando añadimos más capas y más nodos a nuestra red neuronal. Y seguimos sin tener en cuenta los sesgos y la función de activación. Con solo una capa, es relativamente fácil decir que el coste depende de los pesos de $a_1^{(2)}$. Pero en una red neuronal como la que encontramos en la figura 19, es más complicado decir que el coste varía en función de $w_{1,1}$:

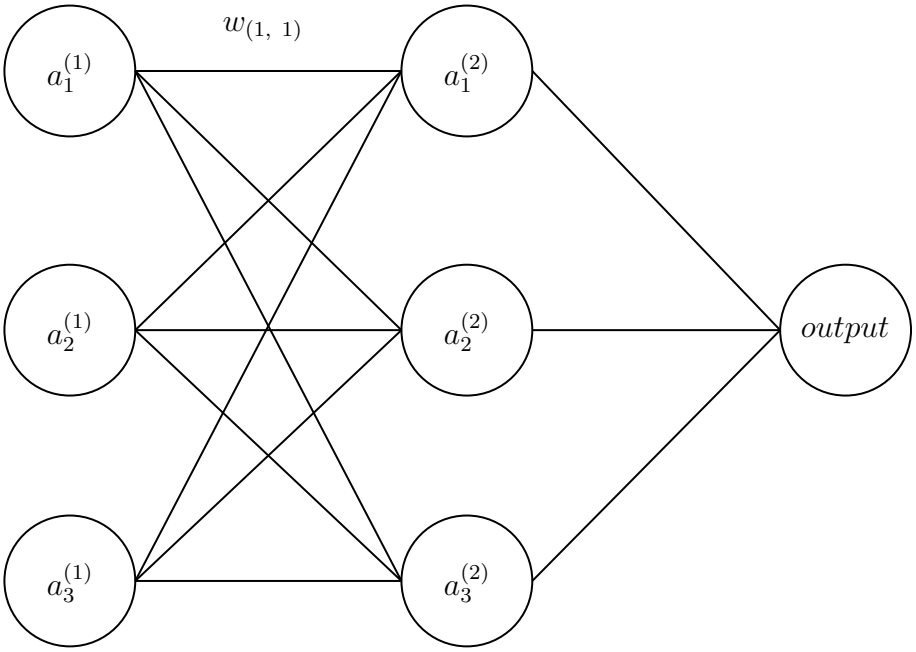


Figura 19: Red neuronal con múltiples capas con múltiples nodos

En este caso, el output, depende de las activaciones de los nodos anteriores, que dependen de las activaciones anteriores. Tendríamos que encontrar como $w_{1,1}$ modifica el coste, junto con todos los otros pesos, y sesgos, para obtener el gradiente del coste. Y todo esto teniendo en cuenta la función de activación. En la siguiente sección introduciremos la retropropagación, que nos explica cómo relacionar todos los pesos y sesgos con el coste.

2.4.3. Retropropagación

Como ya hemos visto, lo que buscamos es el gradiente del coste, que nos proporciona la dirección en la que debemos modificar los pesos para reducir el coste. En el ejemplo que hemos visto previamente, no resultaba muy complicado dado que teníamos los pesos en la misma capa. Vamos a ver otro ejemplo, en el que mantenemos los dos pesos, pero esta vez en capas diferentes. Ahora sí que vamos a mantener los sesgos y la función de activación:

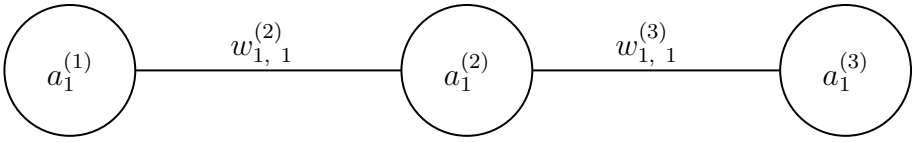


Figura 20: Red neuronal con un solo nodo por capa

Con nuestro sistema de notación anterior no podemos diferenciar entre pesos de diferentes capas, así que añadimos en el lugar del exponente, la capa del nodo que se calcula con el mismo peso, es decir, el peso que se usa para calcular $a_1^{(2)}$ es $w_{1,1}^2$. Este exponente no cumple la función de una potencia, sino que determina la capa del peso.

Para calcular el gradiente del coste, podríamos hacerlo de la misma manera que antes, expresando el coste en función de los dos pesos, y después hacer la derivada parcial del coste con respecto a cada peso. Supongamos que $ValorDeseado(a_1^{(3)})$ es 1, por ejemplo:

$$Cost = (a_1^{(3)} - 1)^2$$
$$Cost = (\sigma(a_1^{(2)} w_{1,1}^{(3)} + b_1^{(3)}) - 1)^2$$
$$Cost = (\sigma(\sigma(a_1^{(1)} w_{1,1}^{(2)} + b_1^{(2)}) w_{1,1}^{(3)} + b_1^{(3)}) - 1)^2$$

De esto, podríamos hacer las derivadas parciales necesarias para calcular el gradiente del coste, y entrenar nuestra red neuronal. Para calcular la derivada de funciones que dependan de otras funciones, podemos usar la regla de la cadena (del inglés *chain rule*).[28]

2.4.3.1 Regla de la cadena

Lo que queremos es hallar el gradiente del coste, y para conseguir esto, necesitamos las derivadas parciales del coste con respecto a todos los pesos y todos los sesgos. Hemos visto que poner todo en una ecuación muy grande, no resulta lo más sencillo, sobre todo si necesitamos miles de pesos y sesgos. Por lo tanto, usamos la regla de la cadena.

$$\nabla Cost = \begin{bmatrix} \frac{\partial Cost}{\partial w_{1,1}^{(2)}} \\ \frac{\partial Cost}{\partial b_1^{(2)}} \\ \frac{\partial Cost}{\partial w_{1,1}^{(3)}} \\ \frac{\partial Cost}{\partial b_1^{(3)}} \end{bmatrix}$$

En el modelo anterior, buscamos un gradiente del coste como este. Cabe recordar que hasta ahora no hemos tenido en cuenta los sesgos para facilitar la explicación, pero también se tienen que tener en cuenta en el coste, para saber como modificarlos.

[28] *Backpropagation*. Jul. de 2023. URL: `https://en.wikipedia.org/wiki/Backpropagation`.

Si encontramos las derivadas parciales necesarias, podemos encontrar la manera en la que debemos modificar los pesos y los sesgos para reducir el coste. Empecemos por $\frac{\partial Cost}{\partial w_{1,1}^{(3)}}$.

Para calcular esta derivada parcial, primero organizamos todas las ecuaciones relevantes.

Sabemos que debemos calcular la media de todos los costes, pero por ahora lo haremos en un caso particular, que lo marcaremos como C_n, donde el subindice n marca la iteración del coste con la que haremos la media. De momento, usamos un subíndice 0 y mantenemos que el valor deseado es 1:

$$C_0 = (a_1^{(3)} - 1)^2$$

Esta ecuación, por sí sola, no nos ayuda mucho, así que vamos a ver de donde sale $a_1^{(3)}$. Esto lo dividiremos en 2 ecuaciones para que más tarde nos sea más fácil derivar la función de activación. Creamos un término medio $z_1^{(3)}$, al que le aplicamos la función de activación para obtener $a_1^{(3)}$:

$$a_1^{(3)} = \sigma(z_1^{(3)})$$
$$z_1^{(3)} = a_1^{(2)} w_{1,1}^{(3)} + b_1^{(3)}$$

Con estas ecuaciones, podemos encontrar diferentes derivadas parciales, por ejemplo con esta última ecuación podemos encontrar $\frac{\partial z_1^{(3)}}{\partial w_{1,1}^{(3)}}$. Con la anterior, podemos encontrar $\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}$, y con la ecuación del coste, podemos encontrar $\frac{\partial C_0}{\partial a_1^{(3)}}$. Y es gracias a la regla de la cadena que podemos encontrar $\frac{\partial C_0}{\partial w_{1,1}^{(3)}}$, que es lo que buscamos[29]:

$$\frac{\partial C_0}{\partial w_{1,1}^{(3)}} = \frac{\partial z_1^{(3)}}{\partial w_{1,1}^{(3)}} \frac{\partial a_1^{(3)}}{\partial z_1^{(3)}} \frac{\partial C_0}{\partial a_1^{(3)}}$$

Para calcular $\frac{\partial C_0}{\partial b_1^{(3)}}$, es lo mismo, sin mayor dificultad, porque estamos trabajando en la capa más próxima al output. Esencialmente, estamos haciendo lo mismo que cuando no usábamos la regla de la cadena. Esto cambia cuando buscamos la derivada del coste con respecto a los pesos o sesgos que se alejen más del output, por ejemplo $\frac{\partial C_0}{\partial w_{1,1}^{(2)}}$.

[29] *Chail rule*. Jun. de 2023. URL: `https://en.wikipedia.org/wiki/Chain_rule`.

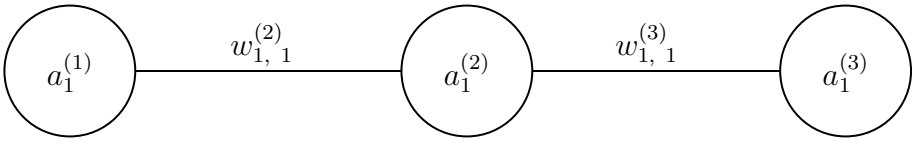


Figura 20: Red neuronal con un solo nodo por capa (repetida de la página 37)

Como $w_{1,1}^{(2)}$ precede a $w_{1,1}^{(3)}$, podemos usar las ecuaciones anteriores y expandirlas:

$$C_0 = (a_1^{(3)} - 1)^2$$
$$a_1^{(3)} = \sigma(z_1^{(3)})$$
$$z_1^{(3)} = a_1^{(2)} w_{1,1}^{(3)} + b_1^{(3)}$$

Igual que con $a_1^{(3)}$, podemos dividir $a_1^{(2)}$ en dos ecuaciones, haciendo uso del término medio $z_1^{(2)}$:

$$a_1^{(2)} = \sigma(z_1^{(2)})$$
$$z_1^{(2)} = a_1^{(1)} w_{1,1}^{(2)} + b_1^{(2)}$$

Con esto, podemos usar la regla de la cadena para calcular $\frac{\partial C_0}{\partial w_{1,1}^{(2)}}$.

$$\frac{\partial C_0}{\partial w_{1,1}^{(2)}} = \frac{\partial z_1^{(2)}}{\partial w_{1,1}^{(2)}} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(3)}}{\partial a_1^{(2)}} \frac{\partial a_1^{(3)}}{\partial z_1^{(3)}} \frac{\partial C_0}{\partial a_1^{(3)}}$$

Vemos que la cadena que se forma es una extensión de la cadena para $\frac{\partial C_0}{\partial w_{1,1}^{(3)}}$, es decir, que la expandimos para adentrarnos más en la red neuronal. Cabe destacar que nos adentramos de derecha a izquierda, dado que el coste se calcula con el nodo de output, de forma que cuanto más se alejen los pesos o sesgos del output, más larga será la cadena.

Pero antes de continuar, vamos a calcular todas estas derivadas parciales. Empezaremos por la primera cadena que hemos formado:

$$\frac{\partial C_0}{\partial w_{1,1}^{(3)}} = \frac{\partial z_1^{(3)}}{\partial w_{1,1}^{(3)}} \frac{\partial a_1^{(3)}}{\partial z_1^{(3)}} \frac{\partial C_0}{\partial a_1^{(3)}}$$

De $z_1^{(3)} = a_1^{(2)} w_{1,1}^{(3)} + b_1^{(3)}$ obtenemos la siguiente derivada parcial:

$$\frac{\partial z_1^{(3)}}{\partial w_{1,1}^{(2)}} = a_1^{(2)}$$

De $a_1^{(3)} = \sigma(z_1^{(3)})$ obtenemos la siguiente derivada parcial, que depende de la función de activación que usemos.

$$\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}} = \sigma'(z_1^{(3)})$$

Para la función sigmoide como función de activación, su derivada parcial es la siguiente:[30]

$$\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}} = \sigma(z_1^{(3)})(1 - \sigma(z_1^{(3)})) = a_1^{(3)}(1 - a_1^{(3)})$$

Y de $C_0 = (a_1^{(3)} - 1)^2$ obtenemos la siguiente derivada parcial:

$$\frac{\partial C_0}{\partial a_1^{(3)}} = 2(a_1^{(3)} - 1)$$

Con todo esto obtenemos la siguiente derivada parcial:

$$\frac{\partial C_0}{\partial w_{1,1}^{(3)}} = a_1^{(2)} a_1^{(3)}(1 - a_1^{(3)})2(a_1^{(3)} - 1)$$

En esta derivada parcial, si sustituimos las incógnitas por sus valores, obtendríamos un valor por el cual deberíamos modificar $w_{1,1}^{(3)}$ en este caso. Es decir, en el caso donde obtengamos un valor de 2 de esta derivada parcial, querrá decir que debemos aumentar en 2 unidades el peso $w_{1,1}^{(3)}$ para que el coste se reduzca.

Como ya hemos visto antes, podemos usar el mismo método para calcular la derivada parcial del coste con respecto al sesgo. En vez de buscar $\frac{\partial C_0}{\partial w_{1,1}^{(3)}}$, buscaríamos $\frac{\partial C_0}{\partial b_1^{(3)}}$. Por lo tanto, en la cadena usaríamos $\frac{\partial z_1^{(3)}}{\partial b_1^{(3)}}$. Pero aparte de esto, es exactamente lo mismo.

Aun así, hemos visto que podemos hacer una de las cadenas que hemos encontrado, pero tenemos que ver si podemos calcular la segunda cadena que hemos encontrado para calcular $\frac{\partial C_0}{\partial w_{1,1}^{(2)}}$:

$$\frac{\partial C_0}{\partial w_{1,1}^{(2)}} = \frac{\partial z_1^{(2)}}{\partial w_{1,1}^{(2)}} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(3)}}{\partial a_1^{(2)}} \frac{\partial a_1^{(3)}}{\partial z_1^{(3)}} \frac{\partial C_0}{\partial a_1^{(3)}}$$

Vemos que las primeras 2 derivadas parciales son las mismas que en la cadena anterior, pero con variables diferentes:

[30] *Activation function.*

De $z_1^{(2)} = a_1^{(1)} w_{1,1}^{(2)} + b_1^{(2)}$, obtenemos la siguiente derivada parcial:

$$\frac{\partial z_1^{(2)}}{\partial w_{1,1}^{(2)}} = a_1^{(1)}$$

De $a_1^{(2)} = \sigma(z_1^{(2)})$, obtenemos la siguiente derivada parcial, que depende de la función de activación que usemos.

$$\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} = \sigma'(z_1^{(2)})$$

Y como ya hemos visto, para la función sigmoide, su derivada parcial es la siguiente:

$$\frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} = \sigma(z_1^{(2)})(1 - \sigma(z_1^{(2)})) = a_1^{(2)}(1 - a_1^{(2)})$$

De $z_1^{(3)} = a_1^{(2)} w_{1,1}^{(3)} + b_1^{(3)}$, obtenemos la siguiente derivada parcial:

$$\frac{\partial z_1^{(3)}}{\partial a_1^{(2)}} = w_{1,1}^{(3)}$$

Vemos que las últimas dos derivadas parciales de esta cadena son iguales a la cadena anterior, así que no tenemos que volver a calcularlas.

$$\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}} = \sigma(z_1^{(3)})(1 - \sigma(z_1^{(3)})) = a_1^{(3)}(1 - a_1^{(3)})$$

$$\frac{\partial C_0}{\partial a_1^{(3)}} = 2(a_1^{(3)} - 1)$$

Y con todo esto obtenemos la siguiente derivada parcial:

$$\frac{\partial C_0}{\partial w_{1,1}^{(2)}} = a_1^{(1)} a_1^{(2)}(1 - a_1^{(2)}) w_{1,1}^{(3)} a_1^{(3)}(1 - a_1^{(3)}) 2(a_1^{(3)} - 1)$$

Vemos que podemos adentrarnos en la red neuronal para calcular la derivada del coste con respecto a cualquier peso o sesgo. Este proceso de adentramiento, es lo que se conoce como retropropagación, y podemos repetirlo hasta el peso o sesgo deseado.[31]

Pero estas derivadas parciales que acabamos de calcular no podemos usarlas directamente para el gradiente del coste. Hemos visto que debemos hacer una media del coste, para calcular el gradiente, ¿Pero qué quiere decir eso?

[31] *Backpropagation.*

Como ya hemos visto, hasta ahora estábamos calculando las derivadas de C_0, es decir, para un solo caso.

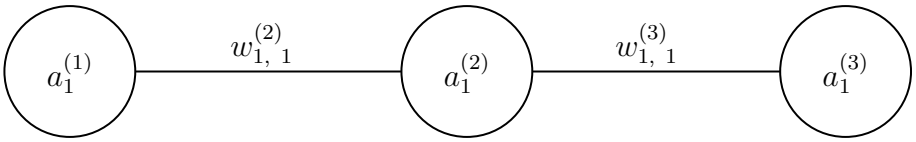


Figura 20: Red neuronal con un solo nodo por capa (repetida de la página 37)

Volviendo al modelo con el que estamos trabajando, pongamos, por ejemplo, que la red neuronal aproxime el valor de la entrada, a 0 o 1. Es decir, si en $a_1^{(1)}$ damos el valor inicial de 0,5 esperamos una activación de 1 en $a_1^{(3)}$

A partir de este caso particular, cuando $a_1^{(1)}$ es 0,5 , podríamos calcular el gradiente del coste. Pero este coste no sería representativo, porque solo nos indicaría como debemos modificar los pesos para que el output devuelva 1.

Es decir, con una entrada de 0,5, el valor deseado es 1, así que el coste se calcula en función de como de lejos de 1 está el output. Como en este caso, el coste es mínimo cuando el output es 1, si entrenamos la red neuronal solo cuando la entrada es 0,5 , estaríamos modificando los pesos y sesgos para que devuelvan un output que tiende a 1. Para este caso, donde el valor deseado es 1, no sería algo malo. Pero si tuviéramos un input de 0,2, el valor deseado sería 0, de forma que el gradiente que hemos calculado en este caso, nos aleja del valor deseado.

Para solucionar esto, debemos tener en cuenta todos los casos posibles, los valores entre 0 y 1, para que el gradiente sea adecuado en todos los casos. Esto se obtiene haciendo una media de todos los gradientes.

Por ejemplo, en el caso donde el input es 0,5 , calculamos el gradiente de este caso obteniendo $-\nabla C_0$. Luego calcularíamos el gradiente del coste con otro caso, por ejemplo con un input de 0,2 , obteniendo $-\nabla C_1$.

Estos gradientes no son más que una lista de números, que nos indican como tenemos que modificar los pesos y sesgos para que, en cada caso particular, se reduzca el coste.

$$-\nabla C_0 = \begin{bmatrix} 3{,}02 \\ 2{,}82 \\ -1{,}64 \\ 1{,}93 \end{bmatrix} \qquad -\nabla C_1 = \begin{bmatrix} -0{,}35 \\ 1{,}53 \\ -1{,}94 \\ 2{,}67 \end{bmatrix}$$

En este caso, con datos inventados, vemos que para C_0, debemos aumentar $w_{1,1}^{(2)}$ en 3,02 , mientras que para C_1, debemos reducir $w_{1,1}^{(3)}$ en 0,35. Estos datos

son diferentes, por lo que acabamos de ver, el gradiente de C_0 busca un valor deseado de 1, mientras que el gradiente de C_1 busca un valor deseado de 0. Para entrenar la red neuronal de forma general, debemos hacer una media de todos los casos. Si solo tenemos en cuenta estos dos casos, haríamos lo siguiente:

$$-\nabla Cost = \begin{bmatrix} \frac{3,02-0,35}{2} \\ \frac{2,82+1,53}{2} \\ \frac{-1,64-1,94}{2} \\ \frac{1,93+2,67}{2} \end{bmatrix} = \begin{bmatrix} 1,335 \\ 2,175 \\ -1,79 \\ 2,3 \end{bmatrix}$$

Está claro que esto se tendría que hacer con todos los casos posibles, no con solo dos. Pero si se llegase a hacer con todos los casos, obtendríamos los valores con los que debemos modificar los pesos y los sesgos para que reduzcan el coste en general, para todos los casos.

Pero de momento, solo estamos trabajando con capas de un nodo cada una. Por ahora hemos visto cómo podemos obtener las diferentes derivadas parciales del coste respecto a cualquier peso. Esto se complica cuando el coste depende de múltiples nodos.

2.4.3.2 Regla de la cadena aplicada a multiples nodos

Pongamos una red neuronal como la siguiente, donde usamos capas con más de un nodo. Pongamos que estamos calculando el gradiente del coste, y queremos encontrar el peso $w^{(3)}_{1,1}$:

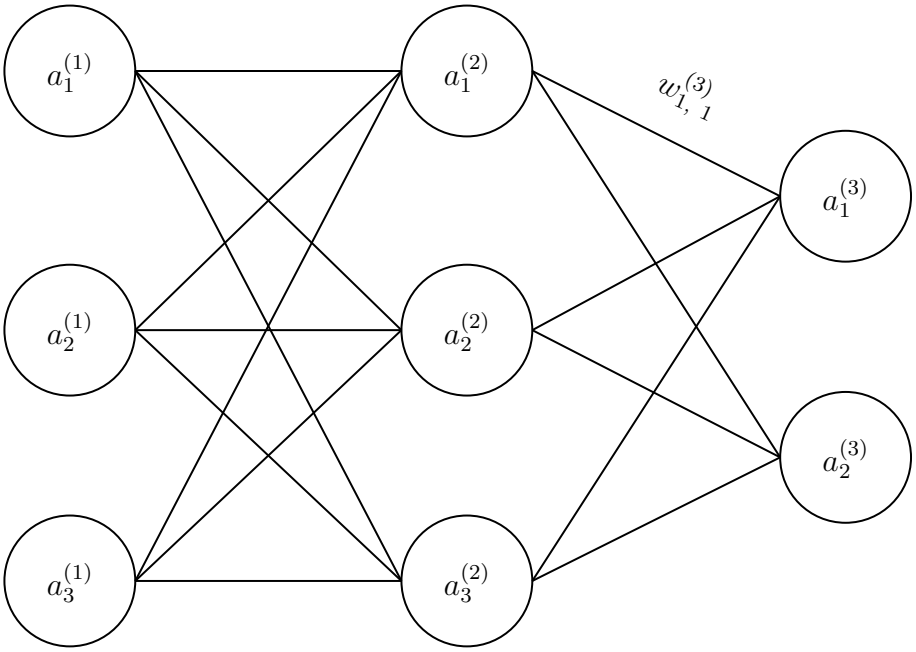


Figura 21: Red neuronal con múltiples capas resaltando $w_{1,1}^{(3)}$

Como el peso con el que estamos trabajando está en la última capa, podemos calcular la derivada del coste con respecto a este peso exactamente igual que antes:

$$\frac{\partial C_0}{\partial w_{1,1}^{(3)}} = \frac{\partial z_1^{(3)}}{\partial w_{1,1}^{(3)}} \frac{\partial a_1^{(3)}}{\partial z_1^{(3)}} \frac{\partial C_0}{\partial a_1^{(3)}}$$

Podemos hacer lo mismo para el resto de pesos y sesgos de esta capa, pero cambiando las derivadas parciales necesarias de la cadena. La cosa se complica cuando intentamos calcular la derivada parcial del coste con respecto a un peso o sesgo, que se adentre más en nuestra red neuronal. Esta complicación es causada por el hecho de que el coste se obtiene a partir de dos nodos. Pongamos que queremos calcular la derivada del coste con respecto a $w_{2,2}^{(2)}$, marcado en rojo, para que se vea mejor:

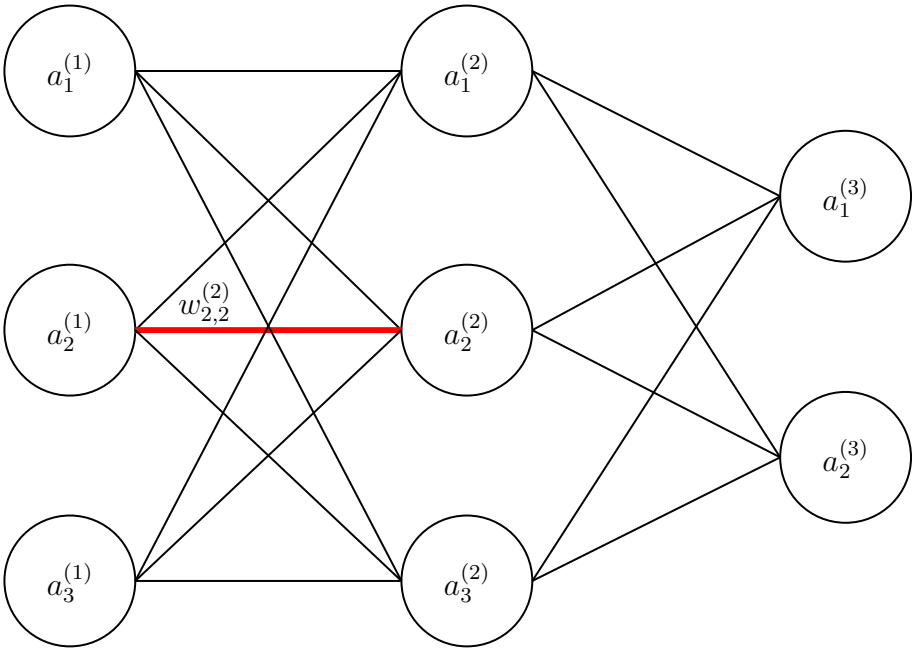


Figura 22: Red neuronal con múltiples capas resaltando $w_{2,2}^{(2)}$

Con lo que ya sabemos, podríamos calcular la derivada parcial del coste con respecto a $w_{2,2}^{(2)}$ de dos maneras: adentrándonos por $a_1^{(3)}$, o por $a_2^{(3)}$, como se ve en las últimas derivadas parciales de cada cadena, resaltadas en rojo:

$$\frac{\partial C_0}{\partial w_{2,2}^{(2)}} = \frac{\partial z_2^{(2)}}{\partial w_{2,2}^{(2)}} \frac{\partial a_2^{(2)}}{\partial z_2^{(2)}} \frac{\partial \color{red} z_1^{(3)}}{\partial a_2^{(2)}} \frac{\partial \color{red} a_1^{(3)}}{\partial \color{red} z_1^{(3)}} \frac{\partial C_0}{\partial \color{red} a_1^{(3)}}$$

$$\frac{\partial C_0}{\partial w_{2,2}^{(2)}} = \frac{\partial z_2^{(2)}}{\partial w_{2,2}^{(2)}} \frac{\partial a_2^{(2)}}{\partial z_2^{(2)}} \frac{\partial \color{red} z_2^{(3)}}{\partial a_2^{(2)}} \frac{\partial \color{red} a_2^{(3)}}{\partial \color{red} z_2^{(3)}} \frac{\partial C_0}{\partial \color{red} a_2^{(3)}}$$

Aunque utilizamos la misma notación en ambos casos, la primera derivada parcial y la segundo no son equivalentes. La primera hace referencia a la derivada parcial obtenida cuando nos adentramos por $a_1^{(3)}$ (es decir cuando el coste se calcula con $a_1^{(3)}$), y la segunda hace referencia a la derivada parcial obtenida cuando nos adentramos por $a_2^{(3)}$ (es decir cuando el coste se calcula con $a_2^{(3)}$).

Esto no es ideal, porque obtendríamos resultados diferentes. Debemos tener en cuenta que el coste viene determinado tanto por $a_1^{(3)}$, como por $a_2^{(3)}$, de forma que debemos incluir los dos nodos en nuestra cadena. Pero antes de eso, podemos arreglar un poco la cadena, parando en el nodo $a_2^{(2)}$:

$$\frac{\partial C_0}{\partial w_{2,2}^{(2)}} = \frac{\partial z_2^{(2)}}{\partial w_{2,2}^{(2)}} \frac{\partial a_2^{(2)}}{\partial z_2^{(2)}} \frac{\partial C_0}{\partial a_2^{(2)}}$$

En realidad solo estamos apartando el problema, pero podemos ver que es a partir de $a_2^{(2)}$, donde el camino empieza a separarse. Podemos obtener $\frac{\partial C_0}{\partial a_2^{(2)}}$, sumando los dos posibles caminos:[32]

$$\frac{\partial C_0}{\partial a_2^{(2)}} = \frac{\partial z_1^{(3)}}{\partial a_2^{(2)}} \frac{\partial a_1^{(3)}}{\partial z_1^{(3)}} \frac{\partial C_0}{\partial a_1^{(3)}} + \frac{\partial z_2^{(3)}}{\partial a_2^{(2)}} \frac{\partial a_2^{(3)}}{\partial z_2^{(3)}} \frac{\partial C_0}{\partial a_2^{(3)}}$$

De forma que nos queda la siguiente cadena:

$$\frac{\partial C_0}{\partial w_{2,2}^{(2)}} = \frac{\partial z_2^{(2)}}{\partial w_{2,2}^{(2)}} \frac{\partial a_2^{(2)}}{\partial z_2^{(2)}} \left(\frac{\partial z_1^{(3)}}{\partial a_2^{(2)}} \frac{\partial a_1^{(3)}}{\partial z_1^{(3)}} \frac{\partial C_0}{\partial a_1^{(3)}} + \frac{\partial z_2^{(3)}}{\partial a_2^{(2)}} \frac{\partial a_2^{(3)}}{\partial z_2^{(3)}} \frac{\partial C_0}{\partial a_2^{(3)}}\right)$$

Vemos que esto se nos puede descontrolar muy rápidamente, sobre todo teniendo en cuenta que cuanto más nos adentremos en la red neuronal durante la retropropagación, más caminos debemos tener en cuenta. Podemos escribir esta cadena de manera más organizada usando el símbolo delta (δ). De momento, volvamos a un peso con el que sea más fácil calcular la derivada parcial del coste con respecto a él:

$$\frac{\partial C_0}{\partial w_{1,1}^{(3)}} = \frac{\partial z_1^{(3)}}{\partial w_{1,1}^{(3)}} \frac{\partial a_1^{(3)}}{\partial z_1^{(3)}} \frac{\partial C_0}{\partial a_1^{(3)}}$$

Vemos que al estar en la última capa, no tenemos que tener en cuenta los diferentes caminos con $w_{1,1}^{(2)}$. Si substituimos las derivadas parciales vemos que casi todos los términos son $a_1^{(3)}$:

$$\frac{\partial C_0}{\partial w_{1,1}^{(3)}} = a_1^{(2)} a_1^{(3)}(1 - a_1^{(3)})2(a_1^{(3)} - 1)$$

Podemos sustituir el bloque que usa $a_1^{(3)}$ (marcado en rojo) por $\delta_1^{(3)}$. Podemos entender $\delta_1^{(3)}$ como otra $z_1^{(3)}$, no en el sentido de que son valores similares, sino que nos ayudan a calcular otros, y más adelante veremos como nos ayuda a calcular todas las derivadas parciales. Cabe destacar, que de momento, nuestro valor deseado en $\frac{\partial C_0}{\partial a_1^{(3)}}$ sigue siendo 1. Por ahora veamos cómo $\delta_1^{(3)}$ nos ayuda a calcular derivadas parciales de la última capa. Si substituimos el bloque de rojo con $\delta_1^{(3)}$ nos queda la siguiente expresión

[32] Sanderson, *Neural Networks*.

$$\frac{\partial C_0}{\partial w_{1,1}^{(3)}} = a_1^{(2)}\delta_1^{(3)}$$

También podemos hacer lo mismo para los sesgos:

$$\frac{\partial C_0}{\partial b_1^{(3)}} = \frac{\partial z_1^{(3)}}{\partial b_1^{(3)}}\frac{\partial a_1^{(3)}}{\partial z_1^{(3)}}\frac{\partial C_0}{\partial a_1^{(3)}}$$

Si substituimos las derivadas parciales, vemos que obtenemos la misma estructura que podemos substituir por $\delta_1^{(3)}$:

$$\frac{\partial C_0}{\partial b_1^{(3)}} = a_1^{(3)}(1-a_1^{(3)})2(a_1^{(3)}-1)$$

$$\frac{\partial C_0}{\partial b_1^{(3)}} = \delta_1^{(3)}$$

Como la derivada parcial de $\frac{\partial z_1^{(3)}}{\partial b_1^{(3)}} = 1$, vemos que la derivada parcial del coste con respecto a un sesgo, es la δ del nodo del sesgo. En la parte práctica, nos centraremos en determinar con exactitud estas relaciones, pero por ahora vemos que podemos definir la derivada parcial del coste con respecto a un peso como un nodo por δ, o con respecto a un sesgo como δ:

$$\frac{\partial C_0}{\partial w_{1,1}^{(3)}} = a_1^{(2)}\delta_1^{(3)} \qquad \frac{\partial C_0}{\partial b_1^{(3)}} = \delta_1^{(3)}$$

Con esto vemos que podemos definir todas las derivadas parciales necesarias para el gradiente del coste con relativa facilidad, siempre y cuando dispongamos de $\delta_m^{(n)}$ para cada nodo. Esto es en cierta manera esconder el problema, pero disponemos de una ecuación que nos permite encontrar todas las δ:[33]

$$\delta_j^{(L)} = \sum_k \delta_k^{(L+1)} w_{k,j}^{(L+1)} a_j^{(L)}(1-a_j^{(L)})$$

Vemos que en esta ecuación, en realidad hacemos lo mismo que antes, también tenemos que sumar los diferentes caminos por los que se puede llegar al peso o sesgo. La diferencia es que con esta ecuación, podemos calcular de antemano todas las δ, y después calcular las derivadas parciales del coste con respecto a los pesos o sesgos.

[33] Shree K. Nayar. *Backpropagation algorithm*. Jun. de 2021. URL: `https://www.youtube.com/watch?v=sIX_9n-1UbM`.

Podemos verificar que esta ecuación funciona. Volvamos a la cadena tan larga de antes que se dividía en dos:

$$\frac{\partial C_0}{\partial w_{2,2}^{(2)}} = \frac{\partial z_2^{(2)}}{\partial w_{2,2}^{(2)}} \frac{\partial a_2^{(2)}}{\partial z_2^{(2)}} (\frac{\partial z_1^{(3)}}{\partial a_2^{(2)}} \frac{\partial a_1^{(3)}}{\partial z_1^{(3)}} \frac{\partial C_0}{\partial a_1^{(3)}} + \frac{\partial z_2^{(3)}}{\partial a_2^{(2)}} \frac{\partial a_2^{(3)}}{\partial z_2^{(3)}} \frac{\partial C_0}{\partial a_2^{(3)}})$$

Si sustituimos las derivadas parciales, obtenemos lo siguiente:

$$\frac{\partial C_0}{\partial w_{2,2}^{(2)}} = a_2^{(1)} a_2^{(2)} (1 - a_2^{(2)}) [w_{1,2}^{(3)} \ a_1^{(3)}(1 - a_1^{(3)}) 2(a_1^{(3)} - 1) + w_{2,2}^{(3)} \ a_2^{(3)}(1 - a_2^{(3)}) 2(a_2^{(3)} - 1)]$$

Vemos de los bloques marcados en rojo, que cada bloque comparte una activación, es decir, que podemos substituirlos por sus δ:

$$\frac{\partial C_0}{\partial w_{2,2}^{(2)}} = a_2^{(1)} a_2^{(2)} (1 - a_2^{(2)}) [w_{1,2}^{(3)} \ \delta_1^{(3)} + w_{2,2}^{(3)} \ \delta_2^{(3)}]$$

Esta ecuación la podemos reorganizar un poco de la siguiente manera:

$$\frac{\partial C_0}{\partial w_{2,2}^{(2)}} = a_2^{(1)} [\delta_1^{(3)} w_{1,2}^{(3)} \ a_2^{(2)}(1 - a_2^{(2)}) + \delta_2^{(3)} w_{2,2}^{(3)} \ a_2^{(2)}(1 - a_2^{(2)})]$$

Si comparamos con la ecuación de antes que usamos para encontrar δ:

$$\delta_j^{(L)} = \sum_k \delta_k^{(L+1)} w_{k,j}^{(L+1)} a_j^{(L)} (1 - a_j^{(L)})$$

Vemos que podemos sustituir el contenido de los corchetes por $\delta_2^{(2)}$:

$$\frac{\partial C_0}{\partial w_{2,2}^{(2)}} = a_2^{(1)} \delta_2^{(2)}$$

Y como habíamos visto antes, podemos calcular la derivada parcial del coste con respecto a un peso con el producto de una activación por su δ, igual que como lo tenemos ahora. Vemos entonces, que con las δ podemos calcular todas las derivadas parciales necesarias para el gradiente del coste.

Este método lo tendríamos que repetir para todos los datos que tengamos, obteniendo derivadas parciales de C_0, C_1, C_2, etc, haríamos la media de estas derivadas parciales y usaríamos esa media en el gradiente del coste.

$$\frac{\partial C}{\partial w_{2,2}^{(2)}} = \frac{\frac{\partial C_0}{\partial w_{2,2}^{(2)}} + \frac{\partial C_1}{\partial w_{2,2}^{(2)}} \cdots \frac{\partial C_n}{\partial w_{2,2}^{(2)}}}{n}$$

Esto lo repetimos para todos los pesos y sesgos de la red neuronal, y obtenemos el gradiente del coste.

$$\nabla Cost = \begin{bmatrix} \frac{\partial C}{\partial w_{1,1}^{(2)}} \\ \frac{\partial C}{\partial b_1^{(2)}} \\ \frac{\partial C}{\partial w_{2,2}^{(2)}} \\ \vdots \\ \frac{\partial C}{\partial b_2^{(3)}} \end{bmatrix}$$

Con el gradiente del coste, sabríamos cómo modificar los pesos y sesgos para mejorar la precisión de la red neuronal poco a poco. Repetiríamos este proceso hasta tener la precisión deseada en nuestra red neuronal.

Con todo lo que hemos visto ya sabemos cómo funciona una red neuronal.

1. Introducimos información a una red neuronal que no ha sido entrenada
2. Con la respuesta de la red neuronal, calculamos todos los δ
3. Con los δ, calculamos la derivada parcial del coste con respecto a cada uno de los pesos y sesgos de nuestra red neuronal
4. Con las derivadas parciales, obtenemos el gradiente del coste (∇C_0) que nos indica cómo debemos modificar los pesos y sesgos.
5. Repetimos todo este proceso con todos los datos a nuestra disposición para hacer una media con todos los gradientes ($\nabla Cost$)
6. Modificamos cada peso y sesgo según indica el negativo del gradiente del coste, para obtener un coste mínimo.
7. Repetimos este proceso hasta tener la precisión deseada

En realidad habría que tener en cuenta un par de cosas más. Por ejemplo que usamos un múltiplo del gradiente porque su magnitud no nos importa o que dosificamos los datos de manera diferente, pero ahora sabemos cómo funciona de manera general una red neuronal. Estos pequeños detalles, los exploraremos a fondo en la parte práctica.

3. Marco Práctico

Con todo lo que acabamos de exponer en el marco teórico, ya sabemos cómo funciona una red neuronal. Sin embargo, para entender realmente cómo funciona lo mejor es pasar a la parte práctica y programar explícitamente una red neuronal. Este será el objetivo del marco práctico.

Para programar nuestra red neuronal usaremos el lenguaje de programación Python. Python es el lenguaje de programación con el que me siento más cómodo, y es por esto que lo usaremos para programar la red neuronal. Además, existen librerías o módulos, que nos ayudarán a la hora de crear nuestra red neuronal. Estas son archivos que contienen funciones y definiciones, que nos ahorran trabajo a la hora de programar.[34] Por ejemplo, usaremos una librería llamada *Matplotlib*, para mostrar las imágenes que pasamos a nuestra red neuronal.

Con relación a las librerías, existen librerías como *TensorFlow*, que sirven para programar redes neuronales mucho más complejas que las que analizamos en este trabajo. Estas librerías te permiten programar una red neuronal sin tener que aprender todas las matemáticas necesarias.[35] Usando estas librerías, podríamos programar una red neuronal con pocas líneas de código,[36] pero con el objetivo final de aprender en profundidad cómo funcionan las redes neuronales, no usaremos estas librerías tan extensas, sino que implementaremos la red neuronal en su totalidad.

3.1. Objetivos de nuestra red neuronal

Muchas veces, obtener los datos para entrenar una red neuronal puede ser tan complicado como programar la red neuronal en sí.[37] Por suerte, existen bases de datos públicas que se pueden usar para entrenar redes neuronales. Una de ellas es la base de datos MNIST, que contiene, 70000 imágenes de dígitos escritos a mano, como la siguiente:

[34] *6. Modules.* URL: `https://docs.python.org/3/tutorial/modules.html`.

[35] *TensorFlow about page.* URL: `https://www.tensorflow.org/about`.

[36] AssemblyAI. *How to create your first neural network with tensorflow!* Feb. de 2022. URL: `https://www.youtube.com/shorts/aUlYe0Rka1o`.

[37] Saheli Roy Choudhury. *Building an A.I. program is easy. the hard part comes after.* Jun. de 2018. URL: `https://www.cnbc.com/2018/06/08/building-an-ai-program-is-easy-the-hard-part-comes-after.html`.

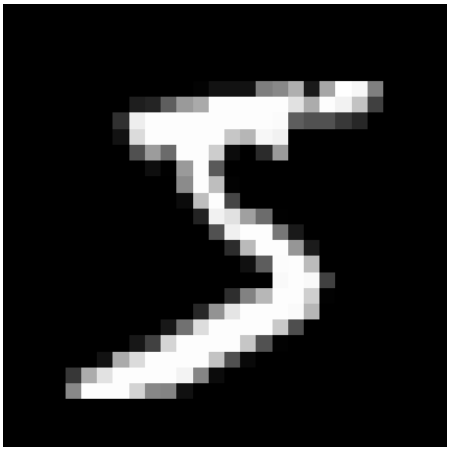


Figura 23: Imagen de un 5 de la base de datos MNIST

Estas imágenes son todas en blanco y negro, con una resolución de 28 por 28 píxeles. Estos dígitos, como humanos, son muy fáciles de identificar. Pero esto se complica mucho para un ordenador. ¿Cómo debemos programar un ordenador para que detecte los diferentes trazados, que forman diferentes números?

En vez de programar un algoritmo que detecte bordes, y que analiza las intersecciones entre bordes, para identificar los dígitos, podemos programar una red neuronal que haga todo esto por nosotros. Este será el objetivo de la parte práctica de este trabajo: programar una red neuronal capaz de identificar dígitos escritos a mano.

3.2. Hagamos una red neuronal

Como ya hemos visto, una red neuronal está compuesta de capas, que están compuestas de nodos. Y en nuestro programa podemos usar clases para crear capas, con las que podemos crear la red neuronal. ¿Pero qué son las clases?

3.2.1. Programación Orientada a Objetos

A la hora de programar, existen diferentes paradigmas que se pueden usar. Uno de ellos es la programación orientada a objetos. En este paradigma, también conocido como OOP (del inglés *Object Oriented Programing*), se usan objetos que pueden tanto guardar información de diversas maneras, como también guardar el código necesario para modificar dichos datos.[38]

[38] Arturo Montejo Ráez y Jiménez Zafra Salud María. "17. Clases y objetos". En: *Curso de Programación Python: 2019*. Anaya Multimedia, 2019.

Esto tiene muchos usos, por ejemplo, podríamos guardar la información de un perro, el nombre, la raza, la edad, etc.

```
class Perro:
    def __init__(self, nombre, raza, edad):
        self.nombre = nombre
        self.raza = raza
        self.edad = edad

# Creamos un objeto usando la clase perro:
perro1 = Perro("Pluto", "labrador", "6")
```

Con este objeto, podríamos acceder al nombre usando `perro1.nombre` . A las clases, podemos añadir métodos, que modifican nuestro objeto.[39] Por ejemplo, podemos añadir un método para obtener la edad en años humanos del perro:

```
class Perro:
    def __init__(self, nombre, raza, edad):
        self.nombre = nombre
        self.raza = raza
        self.edad = edad

    # Definimos un método
    def edadHumana(self):
        # Asumiendo que 1 año de perro son 7 años humanos
        return self.edad * 7

# Creamos un objeto usando la clase perro:
perro1 = Perro("Pluto", "labrador", "6")

print(perro1.edadHumana())
```

Vemos en la línea 15, que llamamos al método `edadHumana()`, y por lo tanto nos devuelve la edad humana del perro.

Esto podemos aplicarlo a nuestra red neuronal. En vez de crear variables para cada peso de cada capa, podemos crear una clase `Layer` (capa en inglés), en la que tengamos los pesos y sesgos de forma más organizada. Podemos también añadir métodos que nos ayuden a manipular los pesos y sesgos, por ejemplo, cuando calculemos el coste o durante la retropropagación.

[39] Ráez y María, "17. Clases y objetos".

3.2.2. Estructura de nuestra red neuronal

Ahora que sabemos cómo funcionan las clases, vamos a ver cómo vamos a estructurar nuestra red neuronal. En este trabajo programaremos una red neuronal de 4 capas. La primera de 784 nodos, un nodo por cada píxel de la imagen. Las dos capas ocultas de 20 nodos. Y la capa final de 10 nodos, donde el nodo más activo es el resultado final. Es decir, si el nodo en la posición 2 es el que está más activado, podemos interpretarlo como que la red neuronal cree que la imagen que le hemos dado es un 2.

La estructura de nuestra red neuronal se ve motivada por diversos factores. Primero de todo, el salto que hacemos de una capa con 784 nodos a una con 20 porque si no habria un exceso de conexiones, la red neuronal no aprende, sino que memoriza los datos de entrenamiento, y ante imágenes nuevas, no sabría qué responder.[40]

En el código, obtenemos una estructura como esta usando 2 clases: la clase `Layer` (capa en inglés), y la clase `Network` (red en inglés), donde unimos 4 objetos `Layer` (en realidad son solo 3 objetos `Layer`, más adelante veremos por qué) para que con la ayuda de métodos de la clase `Network`, podamos tratar a la red neuronal como un solo objeto, y no tener que organizar 4 capas diferentes.

3.2.2.1 Clase Layer

A partir de ahora, empezaremos a ver el código que he desarrollado para este trabajo. Este se puede encontrar en su totalidad en GitHub[41] , pero por ahora iremos viendo diferentes partes, empezando por la clase `Layer`.

En una capa, buscamos:

1. Una forma de organizar todos los pesos y sesgos

2. Funciones que nos ayuden a trabajar con estos datos

Para trabajar con los pesos y sesgos, usaremos la librería `Numpy`,[42] que nos ayuda mucho con multiplicación de matrices, y para generar valores aleatorios para los pesos y sesgos:

[40] *What is overfitting?* URL: `https://www.ibm.com/topics/overfitting`.

[41] Russell. *Código de la red neuronal*. Sep. de 2023. URL: `https://github.com/davoriols/NeuralNetworkTR`.

[42] *Numpy Homepage*. URL: `https://numpy.org/`.

```python
import numpy as np

class Layer:
    # generates the weights and biases of the layer
    def __init__(self, inputSize, outputSize):
        self.weights = np.random.uniform(-0.5, 0.5, (outputSize,
        ↪  inputSize))
        self.biases = np.random.uniform(-0.5, 0.5,(outputSize))
        self.outputSize = outputSize
        self.inputSize = inputSize
```

Con el método `np.random.uniform()`, podemos generar matrices del tamaño que sea. Usamos esto para generar una matriz con los pesos y sesgos necesarios para una capa. Vemos como al generar una instancia de la clase, nos pide un tamaño de entrada (`inputSize`), y un tamaño de salida (`outputSize`). Si tuviésemos una red neuronal como la siguiente:

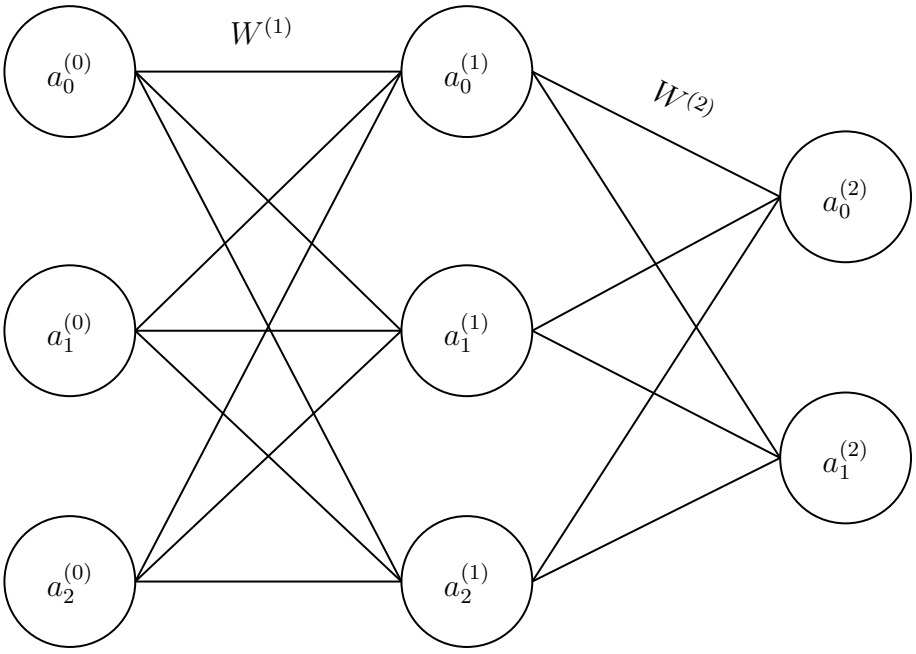


Figura 24: Ejemplo de red neuronal para ilustrar el tamaño de $W^{(1)}$ y $W^{(2)}$

Podríamos calcular las matrices $W^{(1)}$ y $W^{(2)}$ con la siguiente función:

```python
import numpy as np

W1 = np.random.uniform(-0.5, 0.5, (3, 3))

W2 = np.random.uniform(-0.5, 0.5, (2, 3))
```

Donde el -0.5 y 0.5 es el rango de los valores de la matriz, y (3, 3), o (2, 3) es el tamaño de la matriz que queremos. Obteniendo las siguientes matrices, donde cada valor es entre -0.5 y 0.5:

$$W1 = \begin{bmatrix} w_{0,0} & w_{0,1} & w_{0,2} \\ w_{1,0} & w_{1,1} & w_{1,2} \\ w_{2,0} & w_{2,1} & w_{2,2} \end{bmatrix} \quad W2 = \begin{bmatrix} w_{0,0} & w_{0,1} & w_{0,2} \\ w_{1,0} & w_{1,1} & w_{1,2} \end{bmatrix}$$

Esto es lo que hacemos cuando generamos `self.weights` y `self.biases`, solo que con `self.biases`, no hace falta una matriz, y solo generamos un vector con el mismo número de nodos que la capa. Es decir, generamos un vector de tamaño `outputSize`.

Antes de continuar habría que aclarar varias cosas.

1. Primero de todo, estamos empezando a contar desde el 0 y no el 1. Esto es porque es mucho más fácil programarlo así, dado que en Python y muchos otros lenguajes de programación, el primer elemento de una lista siempre es el índice 0. Y en una matriz, el primer elemento es el (0, 0).

2. Otra cosa importante que debemos tener en cuenta, es que los pesos pertenecen a la capa siguiente. Es decir, organizamos los pesos de forma que cada capa tenga los pesos que determinan su activación. De esta forma, la capa 0 ($a^{(0)}$), no tiene pesos. Esto nos facilita mucho el proceso de retropropagación.

Con todo esto, ya tenemos una forma de organizar todos los pesos y sesgos de una capa, pero aún nos queda ver qué métodos utilizamos dentro de la clase `Layer`. Disponemos de solo dos métodos en la clase `Layer`, dado que el código para la retorpropagación, lo más complejo, lo veremos en la clase `Network`.

El primero de estos métodos, el `feedLayerForward()`, donde generamos las activaciones de la capa en que estamos, usando las activaciones de la anterior (las activaciones de la capa anterior se pasan por la variable `input`).

```
def feedLayerForward(self, input):
    self.activations = self.sigmoid(
        np.dot(self.weights, input) + self.biases)

    return self.activations
```

Vemos que en esta función, determinamos las activaciones de la capa, de la siguiente forma:

$$activations = sigmoid((weights * inputs) + biases)$$

Donde `activations` e `input` son vectores con las activaciones de las capas (`activations` es equivalente a $a^{(L)}$, e `input` a $a^{(L-1)}$, `biases` un vector con los sesgos de esta capa que habíamos generado antes ($b^{(L-1)}$), y `weights` la matriz que habíamos generado antes ($W^{(1)}$).

Vemos que nos queda la misma ecuación que habíamos visto en la página 20:

$$\begin{bmatrix} a_0^{(1)} \\ a_1^{(1)} \\ \vdots \\ a_n^{(1)} \end{bmatrix} = \sigma(\begin{bmatrix} w_{0,0} & w_{0,1} & \dots & w_{0,n} \\ w_{1,0} & w_{1,1} & \dots & w_{1,n} \\ \vdots & \vdots & \ddots & \vdots \\ w_{p,0} & w_{p,1} & \dots & w_{p,n} \end{bmatrix} * \begin{bmatrix} a_0^{(0)} \\ a_1^{(0)} \\ \vdots \\ a_n^{(0)} \end{bmatrix} + \begin{bmatrix} b_0^{(1)} \\ b_1^{(1)} \\ \vdots \\ b_n^{(1)} \end{bmatrix})$$

$$a^{(1)} = \sigma(W^{(1)} * a^{(0)} + b^{(1)})$$

Es importante mencionar, que esta función solo determina las activaciones de una capa, y se tendría que ejecutar con cada capa. En la clase `Network`, veremos como determinamos todas las activaciones. En el código usamos la función `np.dot()`[43] de la librería NumPy para poder hacer el producto de matrices, pero también usamos el método `sigmoid()`, que será el segundo método definido en la clase `Layer`:

```
def sigmoid(self, x):
    return 1.0 / (1.0 + np.exp(-x))
```

Sabiendo que `np.exp(-x)` es equivalente a e^{-x},[44] vemos que con este método obtenemos nuestra función de activación, equivalente a la función sigmoide:[45]

[43] *numpy.dot - NumPy v1.26 Manual*. URL: `https://numpy.org/doc/stable/reference/generated/numpy.dot.html#numpy.dot`.

[44] *What exactly does numpy.exp() do*. Dic. de 2015. URL: `https://stackoverflow.com/questions/31951980/what-exactly-does-numpy-exp-do#31952102`.

[45] *Activation function*.

$$\sigma = \frac{1}{1 + e^{-x}}$$

Ahora ya hemos visto como organizar los pesos y sesgos de una capa, y qué métodos usamos. Pero en cierta manera estamos apartando el problema, porque ahora tenemos que preocuparnos por organizar todas las capas y de llamar el método `feedLayerForward()` para cada capa. Esto lo solucionamos con la clase `Network`.

3.2.2.2 Clase Network

En esta clase, buscamos organizar todas las instancias de la clase `Layer`, en un solo objeto con el que nos sea más fácil trabajar. También definiremos nuevos métodos para la retropropagación. Pero antes de eso, vamos a ver que datos necesitamos para la capa `Network`, y cómo los organizamos:

```
1  from Layer import Layer
2
3  class Network:
4      def __init__(self, trainInput, trainLabels, learningRate=0.01):
5          # network with four layers of which the first layer is just
           ↪  the input
6          # therefore we don't declare it since it doesn't have weights
           ↪  or biases
7          self.layers = [Layer(784, 20), Layer(20, 20), Layer(20, 10)]
8          self.trainInput = trainInput
9          self.trainLabels = trainLabels
10         self.learningRate = learningRate
```

Vemos que generamos 4 variables cuando creamos un objeto `Network`:

1. Una lista con todas las capas de la red neuronal, usando la clase `Layer` que acabamos de ver (`layers`)

2. Una variable con todas las imágenes necesarias para el entrenamiento (`trainInput`)

3. Una variable con todas las etiquetas de las imágenes (`trainLabels`)

4. Una variable que determinará cómo de rápido queremos que aprenda la red neuronal. Como hemos visto en la parte teórica, solo nos importa la dirección del gradiente del coste y no la magnitud, así que usamos este valor, 0.01 por defecto, para reducir su magnitud, para que no se salte ningún mínimo. (`learningRate`)

Las variables `trainInput` y `trainLabels`, las obtendremos de la librería `MNIST`,[46] que nos proporciona las imágenes con las que entrenaremos a la red neuronal.

Pero algo que salta más a la vista, es que en la variable `layers` donde definimos las capas de nuestra red neuronal, solo definimos 3 capas. Como bien indica el comentario, esto lo hacemos porque las activaciones de la primera capa, son los datos MNIST directamente. Es decir, que la primera capa no necesita ni pesos ni sesgos, y por lo tanto no creamos un objeto `Layer` para ella.

Si miramos los métodos de la clase `Network`, el primero que encontramos es `feedForward()`:

```
def feedForward(self, input):
    self.layers[2].feedLayerForward(
        self.layers[1].feedLayerForward(
            self.layers[0].feedLayerForward(input)
        )
    )

    return self.layers[2].activations
```

Lo que hacemos en este método es llamar al método `feedLayerForward()`, pero para todas las capas.

Como el método `feedLayerForward()` devuelve las activaciones, podemos usar como argumento el método `feedLayerForward()` de la capa anterior. En la primera capa usamos como argumento `input`, que sería la imagen que queremos que analice.

El siguiente método que veremos es `cost()`, que nos devuelve el coste de la red neuronal:

```
def cost(self, image):
    self.costValue = 0

    # Calculate the cost by looping through each activation
    for activationIndex, activation in
    ↪  enumerate(self.layers[-1].activations):
        self.costValue += np.power(
            activation - self.trainLabels[
                image[0]][image[1]][activationIndex], 2)

    return self.costValue
```

En este método, definimos la variable `costValue`, usando una lista defi-

[46] *MNIST Database python package*. URL: `https://pypi.org/project/mnist/`.

nida como `trainLabels`. Esta lista, es una lista de listas, donde para cada imagen de la base de datos `MNIST`, hay una lista de 10 elementos (uno para cada dígito posible), donde el elemento en la posición de la imagen que está analizando la red neuronal, es 1. Si la red neuronal está analizando una imagen con un 5, en otra parte del código, que veremos más adelante, transformaríamos la etiqueta de la imagen para obtener una lista como la siguiente:

$$labelList = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Vemos que de una imagen con un cinco, recibiríamos una lista donde el elemento de la posición 5 (teniendo en cuenta que la primera posición es la posición 0) es un 1. Más adelante veremos el código para esto.

Vemos en el código que para calcular el coste, que lo único que hace es comparar las activaciones de la última capa con la lista `trainLabels` de la imagen con la que estamos trabajando, y la suma de los cuadrados de la diferencia, es el coste, igual que como habíamos visto antes:

$$Cost = \sum_{i=1}^{n} (a_i^{(output)} - ValorDeseado(a_i^{(output)}))^2$$

Cabe mencionar, que para calcular el coste realmente, se debería hacer una media con el coste de todas las imágenes. Pero como el valor concreto coste no influye directamente en la retropropagación, sino sus derivadas, no vamos a preocuparnos por hacer la media. Solo lo usamos para darnos una idea de como va el proceso de entrenamiento. Más adelante veremos como medir la precisión de la red neuronal, donde tampoco usamos el coste.

El resto de métodos, son para la retropropagación de la red neuronal. Antes de verlos, vamos a plantear cómo vamos a estructurar la retropropagación. Lo que buscamos, es el gradiente del coste:

$$\nabla Cost = \begin{bmatrix} \frac{\partial C}{\partial w_{0,0}^{(1)}} \\ \frac{\partial C}{\partial b_0^{(1)}} \\ \frac{\partial C}{\partial w_{0,1}^{(1)}} \\ \vdots \\ \frac{\partial C}{\partial b_n^{(l)}} \end{bmatrix}$$

Este gradiente del coste, lo obtenemos con las derivadas parciales del coste con respecto a todos los pesos y sesgos, que son los valores que nos importan. Es decir, nosotros solo necesitamos saber cuánto tenemos que mover cada peso y sesgo para que baje el coste.

Para calcular todas las derivadas parciales, hemos visto que podemos facilitarnos el trabajo si encontramos de antemano las δ, con la siguiente fórmula si es la última capa, donde L es el índice de la última capa:

$$\delta_n^{(L)} = a_n^{(L)}(1 - a_n^{(L)})2(a_n^{(L)} - ValorDeseado())$$

Y usando la siguiente fórmula para el resto de capas, haciendo el sumatorio de las previas deltas:

$$\delta_j^{(L)} = \sum_k \delta_k^{(L+1)} w_{k,j}^{(L+1)} a_j^{(L)}(1 - a_j^{(L)})$$

Estas dos ecuaciones necesitan o activaciones que obtenemos con el método `feedForward()` de la clase `Network`, o pesos que generamos en la clase `Layer`, de forma que ya tenemos todo lo que necesitamos.

Todo el proceso de retropropagación, lo dividiremos en 3 métodos:

1. `calculateDeltas()`, donde calcularemos los deltas para cada nodo,
2. `calculateGradient()`, donde usamos las deltas para calcular el gradiente del coste.
3. `backpropagation()`, donde cogeremos los valores del gradiente del coste, y los usaremos para modificar los pesos y sesgos de forma acorde.

Empecemos mirando el método `calculateDeltas()`. En este método, debemos tener en cuenta las dos maneras distintas de calcular delta, en función de la capa. Esto lo podemos hacer en un bucle, en el que miramos si estamos en la última capa:

```
def calculateDeltas(self, image):

    deltas = [[], [], []]

    for layerIndex, layer in zip([2, 1, 0], self.layers):
        # calculates the deltas of the last layer as they are
        ↪ calculated differently
        if layerIndex == 2:
            for activationIndex, activation in enumerate(
                self.layers[-1].activations
            ):
                deltas[2].append(
                    activation
                    * (1 - activation)
                    * 2
                    * (
                        activation
```

```
                        - self.trainLabels[
                            image[0]][image[1]][activationIndex]
                    )
                )

```

Usamos `else`, para discernir del resto de capas, y calculamos las δ de la manera alternativa:

```
        else:
            for activationIndex, activation in enumerate(
                self.layers[layerIndex].activations
            ):
                delta = 0
                for neuronIndex, neuron in enumerate(
                    self.layers[layerIndex + 1].activations
                ):
                    delta += (
                        deltas[layerIndex + 1][neuronIndex]
                        * self.layers[layerIndex +
                        ↪  1].weights[neuronIndex][
                            activationIndex
                        ]
                        * activation
                        * (1 - activation)
                    )
                deltas[layerIndex].append(delta)
    return deltas
```

Vemos que asignamos todas las δ a la variable `deltas`. Esta variable es una lista de listas, estructurada de forma que δ_3^2 sea equivalente a `deltas[2][3]`.

Volviendo a los tres métodos para la retropropagación que habíamos definido antes, nos tocaría ver el método `calculateGradient()`. Como habíamos visto antes, con las δ, calcular las derivadas parciales del coste con respecto a los pesos y sesgos, resulta mucho más fácil. Podemos calcularlas de la siguiente manera:

$$\frac{\partial C}{\partial w_{k,j}^{(L)}} = a_j^{(L-1)}\delta_k^{(L)} \qquad \frac{\partial C}{\partial b_k^{(L)}} = \delta_k^{(L)}$$

Otra cosa que debemos tener en cuenta, es que de las tres capas que hemos generado con la clase `Network`, la primera de ellas usará los datos `MNIST` directamente para $a_j^{(L-1)}$, así que esto lo tendremos que tener en cuenta en el código.

```
def calculateGradient(self, image):
    weightGradient = [[], [], []]

    biasGradient = [[], [], []]

    deltas = self.calculateDeltas(image)

    # calculate the amount we have to change the weights:
    for layerIndex, layer in enumerate(self.layers):
        # checks if it is layer 0 because it needs the input as an
        ↪ activation
        if layerIndex == 0:

            imageShaped =
            ↪ self.trainInput[image[0]][image[1]].reshape(1, 784)
            deltasShaped = np.array(deltas[0]).reshape(
                np.array(deltas[0]).shape[0], -1
            )

            # we do the dot product to calculate the gradient
            weightGradient[0] = self.learningRate * np.dot(
                deltasShaped, imageShaped
            )
```

Vemos en el código, que generamos dos variables `weightGradient`, y `biasGradient`, donde guardaremos los valores de las derivadas parciales.

También vemos que llamamos al método `calculateDeltas()` dentro de `calculateGradient`, así que no tenemos que llamarlo separadamente.

Creamos un bucle, para ciclar todas las capas, y en la primera hacemos varias cosas. Creamos 2 variables, `imageShaped` y `deltasShaped`, que las obtenemos usando el método `np.reshape()`[47] para cambiar la forma de la lista con las deltas de la primera capa y de la imagen.

Es necesario esta transformación, porque así obtenemos una matriz cuando hacemos el producto de matrices de estas dos variables, con el método `np.dot()`.[48] Esta matriz que obtenemos es una matriz con las derivadas parciales del coste con respecto a todos los pesos de la primera capa.

Esta matriz, la multiplicamos por la variable `learningRate`. Cuando creamos un objeto `Network`, generamos, con un valor por defecto de 0.01, la variable `learningRate`. Esto lo hacemos, para reducir la magnitud del gradiente del coste, del cual solo nos importa la dirección. Multiplicando esta matriz por `learningRate`, evitamos que en el descenso de gradiente nos

[47] *numpy.reshape - NumPy v1.26 Manual*. URL: `https://numpy.org/doc/stable/reference/generated/numpy.reshape.html`.

[48] *numpy.dot - NumPy v1.26 Manual*.

saltemos un mínimo. Hay que aclarar que cuando multiplicamos la matriz anterior por `learningRate`, lo hacemos de manera que cada valor de la matriz individualmente se vea multiplicado por `learningRate`. La librería NumPy nos facilita mucho este proceso.

La matriz que obtenemos después de todo este proceso la guardamos en `weightGradient[0]`, de forma que la matriz con las derivadas parciales del coste con respecto al resto de pesos se guardarían en `weightGradient[1]` y `weightGradient[2]`. Es decir, obtenemos una lista de matrices. Vemos en el código, cómo en estas dos capas usamos las activaciones anteriores y no los datos `MNIST`.

```
        else:
            # create variables that store the information in the
            ↪  correct shape to do
            # the dot product
            activationsShaped = self.layers[layerIndex -
            ↪  1].activations
            activationsShaped = activationsShaped.reshape(
                1, activationsShaped.shape[0]
            )
            deltasShaped = np.array(deltas[layerIndex]).reshape(
                np.array(deltas[layerIndex]).shape[0], -1
            )

            # we do the dot product to calculate the gradient
            weightGradient[layerIndex] = self.learningRate * np.dot(
                deltasShaped, activationsShaped
            )

```

Vemos que para el resto de capas, en vez de usar `imageShaped`, usamos `activationsShaped`, que se transforma de una forma similar para poder hacer el producto de matrices. Pero aún nos queda por calcular las derivadas parciales del coste con respecto a los sesgos. Como hemos visto en la página 62, podemos calcularla con δ:

$$\frac{\partial C}{\partial b_k^{(L)}} = \delta_k^{(L)}$$

Como la derivada parcial del coste con respecto al sesgo de un nodo es equivalente a la δ del nodo, podemos simplemente asignar a `biasGradient`, la δ de cada nodo:

```
        biasGradient[layerIndex] = self.learningRate *
        ↪  np.array(deltas[layerIndex])
```

Y las variables `weightGradient` y `biasGradient`, las devolvemos al final del método para poder usarlas después:

```
    return np.array(weightGradient, dtype=object), np.array(
        biasGradient, dtype=object
    )
```

Ahora solo nos queda ver el último método que habíamos determinado para la retropropagación, `backpropagation()`. Este es el más fácil de los tres, porque solo cogemos los valores de `weightGradient` y `biasGradient`, y los restamos a los pesos y sesgos correspondientes:

```
def backpropagation(self, weightGradient, biasGradient):
    # loop through the weights and subtract a multiple of the
    ↪  gradient value for
    # each weight
    for layerIndex, layer in enumerate(self.layers):
        for rowIndex, row in enumerate(layer.weights):
            for weightIndex, weight in enumerate(row):
                self.layers[layerIndex].weights[rowIndex][
                    weightIndex
                ] -=
                ↪  weightGradient[layerIndex][rowIndex][weightIndex]

        # loop through the weights and subtract a multiple of the
        ↪  gradient value for
        # each bias
        for biasIndex, bias in enumerate(layer.biases):
            self.layers[layerIndex].biases[biasIndex] -=
            ↪  biasGradient[layerIndex][
                biasIndex
            ]
```

Es importante destacar que `weightGradient` y `biasGradient` los restamos. Esto lo hacemos porque, como hemos visto en la parte teórica, el gradiente del coste, nos daría la dirección de mayor incremento.[49] Como buscamos el coste mínimo, usamos el negativo del gradiente para ir en dirección opuesta, para llegar a un mínimo en lugar de un máximo.

Pero con todo esto, ya tenemos toda la parte teórica pasada a código. Tenemos el método `feedForward()` que pasa las imágenes por la red neuronal, y con los métodos `calculateDeltas()`, `calculateGradient()` y

[49] *Gradient*.

`backpropagation()`, podemos hacer el proceso de retropropagación. Ahora solo nos queda juntarlo todo para crear nuestra red neuronal.

3.2.3. main.py

Como ya hemos visto, hemos creado dos clases: `Layer` y `Network`. Estas clases, para tener todo más organizado, las podemos guardar en archivos diferentes: `Layer.py` y `Network.py` respectivamente. De esta forma, no tenemos todo el código en un solo archivo, donde podemos olvidarnos en qué parte estaba el código para la retropropagación. De esta forma sabemos que el código de una capa está en `Layer.py`, y el código para la clase `Network` está en `Network.py`.

Los archivos tienen la terminación `.py`, al ser archivos Python. de la misma manera que un archivo de texto acaba en `.txt` o una imagen acaba en `.jpeg` o `.png`

Por lo tanto, en `main.py`, es donde tendremos la parte principal del código. Es donde juntaremos todo lo que hemos hecho para hacer la red neuronal. Cabe mencionar, que en `main.py` no definiremos ninguna clase, sino que usaremos las previamente definidas. Así que en el código tendremos que importarlas, junto con las librerías necesarias:

```
1 import numpy as np
2 import mnist
3 import matplotlib
4 import matplotlib.pyplot as pyplot
5 from Network import Network
```

Vemos que importamos `Numpy`, para facilitar el cálculo con matrices, importamos también la base de datos `MNIST` para las imágenes de los dígitos, y por último importamos la librería `Matplotlib`, que la usaremos para mostrar los resultados de la red neuronal, junto con la imagen en cuestión.[50]

No podemos olvidarnos de la clase `Network`, que también la importamos. No hace falta importar la clase `Layer`, porque esta ya la importamos en la clase `Network`.

Si seguimos viendo el código, veremos que transformamos los datos `MNIST`, para que podamos usarlos en nuestro código:

[50] *Matplotlib Homepage*. URL: `https://matplotlib.org/`.

```
# parse the train images from the mnist database
trainImages = np.array(mnist.train_images())
# normalize the pixel values to pass them as activations
trainImages = (trainImages) / 255
# reshape the array to have 600 mini sets of 100 images
trainImages = np.reshape(trainImages, (600, 100, 784))


# parse the test images from the mnist database
testImages = np.array(mnist.test_images())
# normalize the pixel values to pass them as activations
testImages = (testImages) / 255
# reshape the array to have an array of 10000 images
testImages = np.reshape(testImages, (10000, 784))
```

En la base de datos `MNIST`, obtenemos 4 variables. Dos de ellas con imágenes con los dígitos escritos a mano, y las otras dos, que contienen el número que hay en la imagen realmente. Esto también se conoce como etiqueta. Tanto las imágenes como las etiquetas se dividen en dos, para tener un set de 60000 imágenes para entrenar la red neuronal, y otro set de 10000 imágenes, junto con las 60000 etiquetas de las imágenes de entrenamiento y otras 10000 etiquetas de las imágenes restantes. Las 10000 imágenes que no se usan para entrenar la red neuronal, son imágenes de prueba con las que mediremos la precisión de la red neuronal con imágenes que nunca haya visto.

El problema surge en el formato de estas imágenes. Por defecto, son una lista de listas, conteniendo los valores de los píxeles de la foto. Es decir, es una lista que contiene listas de 784 elementos, donde cada elemento es un píxel de la imagen. Pero el valor del píxel es entre 0 y 255, de manera que un píxel negro tendría un valor de 0 y uno completamente blanco un valor de 255.

Como usaremos estas imágenes como las activaciones de la primera capa, necesitamos que tengan valores entre 0 y 1. Esto lo solucionamos dividiendo todos los valores por 255, el máximo valor. Esto lo tenemos que hacer 2 veces, para `trainImages` y para `testImages`, pero por suerte, la librería `Numpy` nos facilita mucho este proceso.

Luego cambiamos la forma de `trainImages` para obtener 600 sets de 100 imágenes cada uno, y de `testImages` para tener una lista con 10000 imágenes. Más adelante veremos por qué creamos 600 sets de imágenes, y no uno con todas las imágenes.

Y de la misma forma que tenemos que cambiar el formato de las imágenes, también tenemos que cambiar el formato de las etiquetas:

```
# parse the labels form the mnist database
labels = np.array(mnist.train_labels())
# create trainLabel list with lists of zeros with the label index being 1
# for label 5, trainLabel would be [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
# do this for every label
trainLabels = []
for label in labels:
    labelList = np.zeros(10)
    labelList[label] = 1
    trainLabels.append(labelList)
# reshape labels array to have 600 mini sets of 100 labels
trainLabels = np.reshape(trainLabels, (600, 100, 10))

# get the test labels
testLabels = mnist.test_labels()
```

Vemos que para `trainLabels`, transformamos la etiqueta a una lista donde el elemento en la posición de la etiqueta sea 1. Como habíamos visto antes en el método `cost()`, de la clase `Network`, necesitamos que las etiquetas estén en este formato, y es aquí donde lo hacemos.

Vemos también que este proceso solo aplica a las etiquetas de `trainLabels` y no `testLabels`. Cuando analicemos los resultados de la red neuronal, usando `testImages` y `testLabels`, no usaremos el coste, así que no necesitamos hacer esta transformación dos veces.

A continuación, vamos a ver el código, donde realmente entrenamos a la red neuronal. Lo veremos entero primero, y lo iremos diseccionando:

```
# generate network object
network = Network(trainImages, trainLabels)

# iterator to know how far is the training process
iteration = 0

for epoch in range(4):
    # Wrap everything in a loop that loops through all the sets
    for minisetIndex, miniset in enumerate(trainImages):
        iteration += 1

        # first iteration of the training
        # done outside the main loop because this way, weightGradientStack and
        # biasGradientStack isn't reasigned for each iteration
        network.feedForward(trainImages[0][0])
        print(f"{iteration} --> {network.cost((0, 0))}")
        print(network.layers[2].activations)
```

```
18
19          # I create a gradientStack variable to hold the gradient
            ↪ values for the weights
20          # and biases
21          # this is done by assigning the variable to weightGradient -
            ↪ weightGradientStack
22          # to get an array with the same shape, where each value is 0,
            ↪ and doesn't affect
23          # the result
24          weightGradient, biasGradient = network.calculateGradient((0,
            ↪ 0))
25
26          weightGradientStack = weightGradient - weightGradient
27          biasGradientStack = biasGradient - biasGradient
28
29          # loops throuhg all the images in the miniset, and calculates
            ↪ the weightGradient
30          # biasGradient, and add them to their stack
31          for imageIndex, image in enumerate(miniset):
32              network.feedForward(image)
33
34              weightGradient, biasGradient = network.calculateGradient(
35                  (minisetIndex, imageIndex)
36              )
37
38              weightGradientStack += weightGradient
39              biasGradientStack += biasGradient
40
41          # once all images in the miniset are trained on, we get the
            ↪ stack, and use
42          # these values for the backpropagation function
43
44          network.backpropagation(weightGradientStack,
            ↪ biasGradientStack)
```

Lo primero que veremos, es que creamos una objeto `network`, de la clase `Network` que hemos definido antes. Pero viendo las siguientes líneas, encontramos algo que no hemos visto antes:

```
1  # iterator to know how far is the training process
2  iteration = 0
3
4  for epoch in range(4):
5      # Wrap everything in a loop that loops through all the sets
6      for minisetIndex, miniset in enumerate(trainImages):
7          iteration += 1
8
```

Definimos una variable `epoch`, y hacemos un bucle que repita 4 veces. ¿Pero qué es un epoch? Los epoch son cada vez que la red neuronal ve todos los datos de entrenamiento. Por lo tanto, si le ponemos que haga 4 epochs, nuestra red neuronal verá un total de 4 veces cada imagen. Es importante no hacer demasiados epochs durante el entrenamiento, porque sino, la red neuronal puede llegar a memorizar los datos de entrenamiento, de manera que luego no sepa identificar datos que nunca haya visto antes.[51]

Pero luego vemos que cicla los minisets que habíamos creado antes. Como habíamos visto en la parte teórica, para conseguir una retropropagación correcta, en teoría deberíamos calcular el gradiente con todas las imágenes, hacer una media, y la media del gradiente usarla para modificar los pesos y los sesgos.

Sin embargo, no resulta computacionalmente eficiente calcular el gradiente de 60000 imágenes distintas, Así que dividimos las imágenes en 600 sets de 100 imágenes. Con las que calculamos el gradiente de las 100 imágenes del set, que es más asequible, y la media de este nuevo gradiente la usamos para modificar los pesos y sesgos.[52]

Esta variación del descenso de gradiente es conocido como descenso de gradiente estocástico (del inglés *stochastic gradient descent*).[53]

Por último, vemos que al principio del bucle definimos `iteration`, una variable que aumentara en 1 cada set que pasemos por la red neuronal. De esta forma podemos determinar con cuantos sets ha entrenado la red neuronal.

```
        network.feedForward(trainImages[0][0])
        print(f"{iteration} --> {network.cost((0, 0))}")
        print(network.layers[2].activations)

        weightGradient, biasGradient = network.calculateGradient((0,
        ↪  0))

        weightGradientStack = weightGradient - weightGradient
        biasGradientStack = biasGradient - biasGradient
```

Si analizamos el código, vemos que llamamos al método `feedForward()` por primera vez, con la primera imagen del primer set, que es una imagen de un cinco que hemos visto en la figura 23:

[51] *What is overfitting?*

[52] Sanderson, *Neural Networks*.

[53] *Stochastic gradient descent*. Ago. de 2023. URL: `https://en.wikipedia.org/wiki/Stochastic_gradient_descent`.

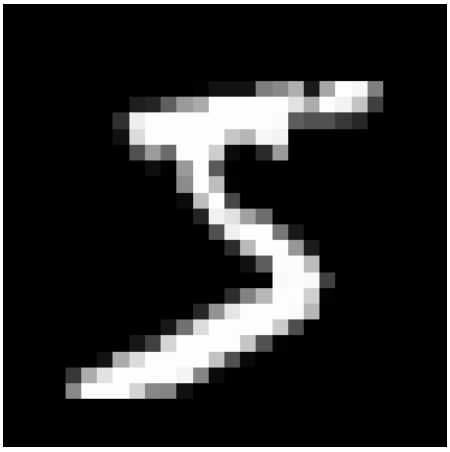


Figura 23: Imágen de un 5 de la base de datos MNIST (repetida de la página 52)

Una vez hecho el `feedForward()`, imprimimos a la terminal el coste que tiene para esta imagen de un 5, e imprimimos también las activaciones de la última capa. Esto lo hacemos para tener una idea de como va aprendiendo nuestra red neuronal mientras la entrenamos. Como es solo para darnos una idea, no nos preocupamos en hacer la media con todos los costes del set de imágenes, e imprimimos a la terminal del coste de la primera imagen.

Pero esto que hacemos, no es para la retropropagación directamente. Las activaciones que generamos con `feedForward()`, que en realidad las generamos con `feedLayerForward()`, podemos usarlas para calcular las variables `weightGradient` y `biasGradient`.

Como hemos visto antes, queremos, en teoría, hacer la media con todos los `weightGradient` y `biasGradient` de un set de imágenes, y la media usarla en el método `backpropagation()`. Por lo tanto, creamos las variables `weightGradientStack` y `biasGradientStack`. Si nos fijamos, estas variables las creamos con la diferencia de `weightGradient` y `biasGradient`. Es decir, estamos diciendo que `weightGradientStack` es igual a `weightGradient` menos `weightGradient`.

Esto a primera vista parece que nos devuelva 0. Sin embargo lo hacemos así, porque de esta forma obtenemos una lista de matrices igual que `weightGradient`, pero gracias a la resta, todos los valores de las matrices son 0. Por lo tanto, a las variables `weightGradientStack` y `biasGradientStack`, podemos simplemente sumarle `weightGradient` y `biasGradient` de cada imagen, y después, en teoría, hacer la media de `weightGradientStack` y `biasGradientStack` para usarlo en el método `backpropagation()`.

```
        for imageIndex, image in enumerate(miniset):
            network.feedForward(image)

            weightGradient, biasGradient = network.calculateGradient(
                (minisetIndex, imageIndex)
            )

            weightGradientStack += weightGradient
            biasGradientStack += biasGradient


        network.backpropagation(weightGradientStack,
        ↪  biasGradientStack)
```

Si seguimos viendo el código de `main.py`, vemos que empezamos un bucle, donde para cada imagen de un set, calculamos `weightGradient` y `biasGradient`, y sumamos estas variables a `weightGradientStack` y `biasGradientStack`.

Pero es en estas líneas de código, que encontramos el error más importante de este trabajo. Como hemos visto, en teoría, debemos hacer la media de `weightGradientStack` y `biasGradientStack`, sin embargo, vemos que cuando llamamos a `backpropagation()`, usamos `weightGradientStack` y `biasGradientStack` directamente sin hacer la media.

Según lo que hemos visto en la parte teórica, deberíamos hacer la media de `weightGradientStack` y `biasGradientStack`, y usar la media en `backpropagation()`. Sin embargo, cuando le programo para que haga la media, la red neuronal no aprende, el coste oscila entre 0,9 y 1,0 que no es nada bueno. Y aunque es verdad que el coste era solo para darnos una idea, con otro programa que veremos más adelante con el que podemos medir la precisión de la red neuronal, veremos que acertará el 12 % de las veces, es decir, no mucho mejor que escogiendo aleatoriamente.

Pero si no hacemos la media y entrenamos la red neuronal, que más adelante veremos algunas dificultades con entrenar la red neuronal, vemos que el coste oscila entre 0,01 y 0,2 , mucho mejor que antes. Y usando el programa para ver la precisión de la red neuronal, veremos que sin hacer la media obtenemos una precisión del 92 %.

Este fallo no es ideal, porque viene con sus desventajas, que el proceso de entrenamiento, no siempre es exitoso, que implica que debemos entrenar múltiples veces nuestra red neuronal para conseguir los pesos y sesgos deseados.

El hecho de que se entrene correctamente solo algunas veces, lleva a pensar que la variable `learningRate`, sea demasiado alta, y que la red neuronal aprende demasiado rápido. `learningRate` es el valor por el que multiplicamos

el gradiente para que no nos afecte su magnitud. Si este valor es demasiado alto, la magnitud del gradiente podría ser demasiado alta, y que se salte un mínimo, que es lo que buscamos.

Como lo que estamos haciendo es multiplicar por 0.01, también podemos entenderlo como dividirlo entre 100, que es lo que necesitamos hacer para realizar la media, es decir que con `learningRate` y estamos haciendo la media, pero eso implica que estaríamos usando un `learningRate` de 1, que comparado con otras redes neuronales similares, es muy alto.[54]

Si hago la media, o aumento `learningRate`, la red neuronal aprende considerablemente más lento que si se deja como está, que mantienen la teoría de que la red neuronal aprende demasiado rápido. Según mi criterio, el error está relacionado con `learningRate`, o que no he usado correctamente la librería `Numpy` para hacer la media con las matrices, que también es otra posibilidad. Sin embargo, cuando se entrena la red neuronal correctamente, adivina los números sorprendentemente bien. Como hemos visto antes, con una precisión del 92 %, más adelante veremos como llegamos a este número.

Si seguimos viendo el código, veremos que guardamos los valores de los pesos y sesgos de la red neuronal ya entrenada. Esto lo hacemos para que podamos usar la red neuronal sin tener que entrenarla cada vez.

```
# export the weights and biases to their own csv, so that the network doesn't have to be trained every time
np.savetxt("data/weights/weights0.csv", network.layers[0].weights, delimiter=",")
np.savetxt("data/weights/weights1.csv", network.layers[1].weights, delimiter=",")
np.savetxt("data/weights/weights2.csv", network.layers[2].weights, delimiter=",")

np.savetxt("data/biases/biases0.csv", network.layers[0].biases, delimiter=",")
np.savetxt("data/biases/biases1.csv", network.layers[1].biases, delimiter=",")
np.savetxt("data/biases/biases2.csv", network.layers[2].biases, delimiter=",")
```

Para no complicarlo mucho, los pesos y sesgos de cada capa están guardados en su propio archivo. los valores están guardados en un archivo `CSV` (del inglés *comma-separated-values*), donde gracias a la librería `Numpy`, podemos importarlos fácilmente en otros programas.

Estos archivos se guardan en la carpeta `data`, en la cual los pesos están en la carpeta `weights`, y los sesgos en la carpeta `biases`.

[54] Academy, *Neural Networks explained from scratch using Python*.

En el último fragmento de `main.py`, vemos que escogemos una imagen de las 10000 imágenes que no ha visto antes, para ver que resultado nos devuelve la red neuronal, y compararlo con la etiqueta de la imagen.

```
# test the network, with the test data, that has not been seen by the network before
while True:
    imageChosen = int(input("choose a test image (0-9999)"))

    # feed forward to get activations
    network.feedForward(testImages[imageChosen])
    print(network.layers[2].activations)
    activations = list(network.layers[2].activations)

    # represent images in matplotlib:
    # reshape the image to a (28, 28) dimension array
    image = np.reshape(testImages[imageChosen], (28, 28))
    # show the image in matplotlib
    pyplot.title(
        f"label says: {testLabels[imageChosen]} \n network says: {activations.index(max(activations))}"
    )
    pyplot.imshow(image, interpolation="nearest", cmap="gray")
    pyplot.show()
```

Tras preguntar al usuario que escoja una imagen, calculamos las activaciones para esa imagen. Gracias a la librería `Matplotlib`, podemos representar gráficamente la imagen escogida, junto con la etiqueta de la imagen y el resultado de la red neuronal.

Obtenemos el resultado de la red neuronal, buscando el elemento de la lista de activaciones más alto, es decir, si el elemento en la posición 4 es el más alto, la red neuronal cree que la imagen es un 4.

Con este código generamos la siguiente interfaz, que nos da la etiqueta de la imagen, el resultado de la red neuronal, y la imagen en sí:

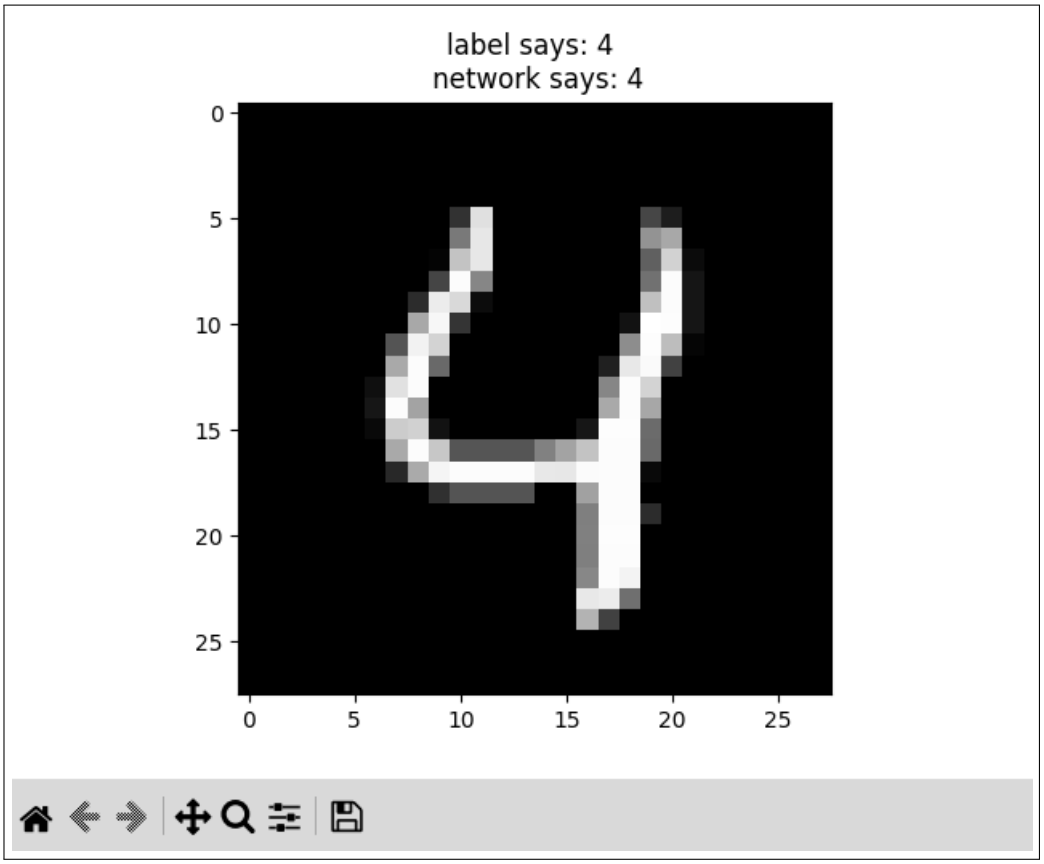


Figura 25: Respuesta de la red neuronal, identificando un 4 en la imágen

Y con eso acabamos de ver todo el código necesario para entrenar una red neuronal. Vemos como tenemos diferentes partes de la teoría en diferentes métodos, que al final juntamos en `main.py`.

3.2.4. test.py

Pero aún nos queda ver como de buena es nuestra red neuronal. Como hemos visto antes, usamos otro programa para ver la precisión de nuestra red neuronal, que será este. En `test.py` buscaremos 2 cosas:

1. Ver el grado de precisión de nuestra red neuronal, y tener la opción de ver donde se equivoca.

2. Poder usar la red neuronal sin tener que entrenarla cada vez. Para esto usaremos el último bloque de código de `main.py`, donde nos pide que seleccionemos una imagen.

Debido a que la red neuronal no siempre se entrena correctamente, añadiremos la opción de usar pesos y sesgos previamente entrenados, para que el usuario no tenga que entrenar la red neuronal, y pueda usar estos valores.

```python
import numpy as np
from numpy import genfromtxt
import mnist
from Network import Network
import matplotlib.pyplot as pyplot

# Configuration options:

# change to True to used saved training data, or False to data from previous training
useTrainData = False

# change to True to show a pyplot of the incorrect gueses, of False to not show
showResults = False
```

Si miramos el código, encontramos que, aparte de importar todas las librerías y clases, definimos dos variables: `useTrainData` y `showResults`. Como indica el comentario, podemos cambiar su valor, para que usemos pesos y sesgos preentrenados, y para mostrar las imágenes en las que la red neuronal se equivoca.

```python
# parse the test images from the mnist database
testImages = np.array(mnist.test_images())
# normalize the pixel values to pass them as activations
testImages = (testImages - np.min(testImages)) / (
    np.max(testImages) - np.min(testImages)
)
# reshape the array to have a list of 10000 images
testImages = np.reshape(testImages, (10000, 784))

# import test labels
testLabels = mnist.test_labels()

# generate network object
# for the methods we use to test, we don't accually need trainImages or trainLabels, but the network
# requires it, so we just pass None
network = Network(None, None)
```

Si continuamos, veremos código similar al de `main.py`, donde importamos los datos de prueba de la base de datos `MNIST`. En este caso, no importamos los datos de entrenamiento, porque la red neuronal ya está entrenada.

Si nos fijamos, vemos que creamos un objeto `network` con la clase `Network`, pero en vez de pasar los datos de entrenamiento pasamos `None`. Esto lo hacemos porque como no vamos a entrenar la red neuronal, no vamos a usar ningún método que necesite estos datos, y por lo tanto no tenemos que importarlos. Eso sí, como bien indica el comentario, pasamos `None`, porque la clase necesita que pasemos algo, porque si no salta un error.

```
# checks which weights and biases values we want to use
if useTrainData:
    # get the pretrained weights
    network.layers[0].weights = genfromtxt(
        "trainedData/weights/weights0.csv", delimiter=",", dtype=None
    )

    network.layers[1].weights = genfromtxt(
        "trainedData/weights/weights1.csv", delimiter=",", dtype=None
    )

    network.layers[2].weights = genfromtxt(
        "trainedData/weights/weights2.csv", delimiter=",", dtype=None
    )

    # do the same for biases
    network.layers[0].biases = genfromtxt(
        "trainedData/biases/biases0.csv", delimiter=",", dtype=None
    )

    network.layers[1].biases = genfromtxt(
        "trainedData/biases/biases1.csv", delimiter=",", dtype=None
    )

    network.layers[2].biases = genfromtxt(
        "trainedData/biases/biases2.csv", delimiter=",", dtype=None
    )

else:
    # get the user-trained weights
    network.layers[0].weights = genfromtxt(
        "data/weights/weights0.csv", delimiter=",", dtype=None
    )

    network.layers[1].weights = genfromtxt(
        "data/weights/weights1.csv", delimiter=",", dtype=None
    )

    network.layers[2].weights = genfromtxt(
        "data/weights/weights2.csv", delimiter=",", dtype=None
```

```
    )

    # do the same for biases
    network.layers[0].biases = genfromtxt(
        "data/biases/biases0.csv", delimiter=",", dtype=None
    )

    network.layers[1].biases = genfromtxt(
        "data/biases/biases1.csv", delimiter=",", dtype=None
    )

    network.layers[2].biases = genfromtxt(
        "data/biases/biases2.csv", delimiter=",", dtype=None
    )
```

Luego, importamos los valores de los pesos y sesgos, pero mirando antes el valore de `useTrainData`, si es `True`, usa valores preentrenados, si no usa los valores que ha generado el usuario. Este proceso nos lo facilita mucho la librería `Numpy` con la función `genfromtxt()`,[55] que a partir de los archivos `CSV` que habíamos generado antes, podemos asignar los valores a variables fácilmente. Pero vamos a ver cómo mide la precisión de la red neuronal `test.py`:

```
totalGuesses = 0
correctGuesses = 0


for imageIndex, image in enumerate(testImages):
    totalGuesses += 1

    network.feedForward(image)

    activations = list(network.layers[2].activations)

    if activations.index(max(activations)) == testLabels[imageIndex]:
        correctGuesses += 1

    else:
        if showResults:
            image = np.reshape(image, (28, 28))
            # show the image in matplotlib
            pyplot.title(
                f"label says: {testLabels[imageIndex]} \n network
                ↪  says: {activations.index(max(activations))}"
```

[55] *numpy.genfromtxt - NumPy v1.26 Manual*. URL: `https://numpy.org/doc/stable/reference/generated/numpy.genfromtxt.html`.

```
            )
            pyplot.imshow(image, interpolation="nearest",
            ↪ cmap="gray")
            pyplot.show()

    print(f"correct guesses: {correctGuesses}")
    print(f"total guessses:  {totalGuesses}")
    print(f"accuracy:        {correctGuesses / totalGuesses * 100}")
```

Primero generamos 2 variables: `totalGuesses` y `correctGuesses`. Lo que hacemos en este fragmento de código, es ver cuantas veces acierta la red neuronal, y lo dividimos por cuantas imágenes lleva. Este valor multiplicado por 100 nos da el porcentaje de éxito de la red neuronal, con imágenes con las que no ha sido entrenada.

Opcionalmente, si la variable `showResults` es `True`, muestra las imágenes en las que la red neuronal se equivoca.

Y si nos fijamos en el último fragmento de `text.py`, encontramos el mismo código que en `main.py`:

```
while True:

    imageChosen = int(input("choose a test image (0-9999)"))

    # feed forward to get activations
    network.feedForward(testImages[imageChosen])
    print(network.layers[2].activations)
    activations = list(network.layers[2].activations)

    # represent images in matplotlib:
    # reshape the image to a (28, 28) shape array
    image = np.reshape(testImages[imageChosen], (28, 28))
    # show the image in matplotlib
    pyplot.title(f"label says: {testLabels[imageChosen]} \n network
    ↪ says: {activations.index(max(activations))}")
    pyplot.imshow(image, interpolation="nearest", cmap="gray")
    pyplot.show()
```

Este fragmento permite al usuario seleccionar cualquiera de las 10000 imágenes que no ha visto la red neuronal, para ver qué resultado da. Repetimos este código, para que no sea necesario entrenar la red neuronal cada vez que queramos usarla. En lugar de ejecutar `main.py` que tarda unos minutos en entrenar la red neuronal, para después poder usarla, podemos ejecutar `test.py`, que solo tarda unos segundos en calcular la precisión, y después nos permite seleccionar una de las imágenes.

3.3. Resultados

Hasta ahora hemos visto cómo funciona una red neuronal y también cómo programarla, pero no hemos testeado su funcionamiento. Se ha mostrado el programa `test.py`, que nos mide la precisión de la red neuronal a la hora de detectar dígitos escritos a mano, así que vamos a ver cómo de precisa es nuestra red neuronal.

Si ejecutamos el programa `test.py`, se imprime en la terminal el número de imágenes que identifica correctamente la red neuronal, el número de imágenes de prueba que ha analizado y el porcentaje de aciertos.

```
correct guesses: 9210
total guessses:  10000
accuracy:        92.10000000000001
choose a test image (0-9999)
```

Figura 26: Captura de la terminal tras ejecutar `test.py`

Tras ejecutar el programa en su totalidad, en este test particular, vemos en la figura 26 que de las 10000 imágenes de prueba que nos ofrece la base de datos MNIST, nuestra red neuronal es capaz de identificar correctamente 9210. Esto nos da un porcentaje de acierto del 92.10 %.

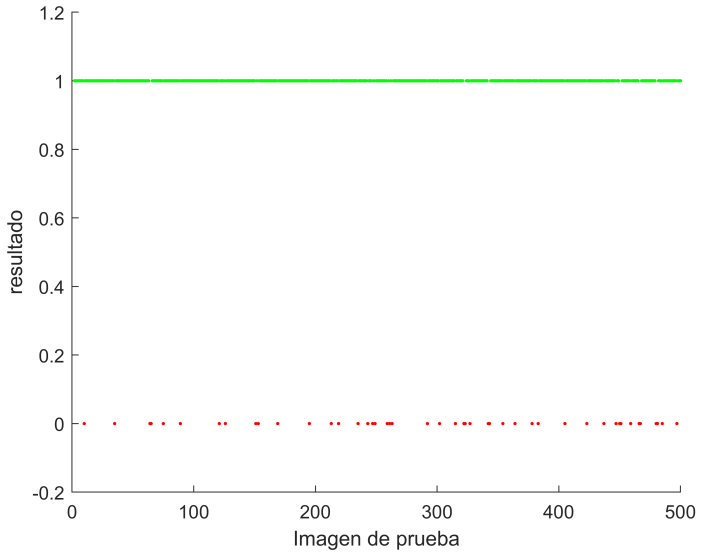


Figura 27: Gráfica con los aciertos y fallos de la red neuronal con las imágenes de prueba

En la figura 27, se muestra los aciertos (indicado en verde, de valor 1) y los errores (indicados en rojo, de valor 0) para las primeras 500 imágenes de prueba. Vemos que la red neuronal ha acertado 453 dígitos de un total de 500, con solo 46 errores. Como los valores iniciales de los pesos y sesgos de la red neuronal son aleatorios, otro entrenamiento puede dar pesos y sesgos diferentes, que resulten en mayor o menor porcentaje de aciertos. .

En este sentido, es importante comprobar si la red neuronal aprende correctamente. En la figura 28 se muestra los valores numéricos de 3 pesos en función del set imágenes con los que se ha entrenado la red neuronal.

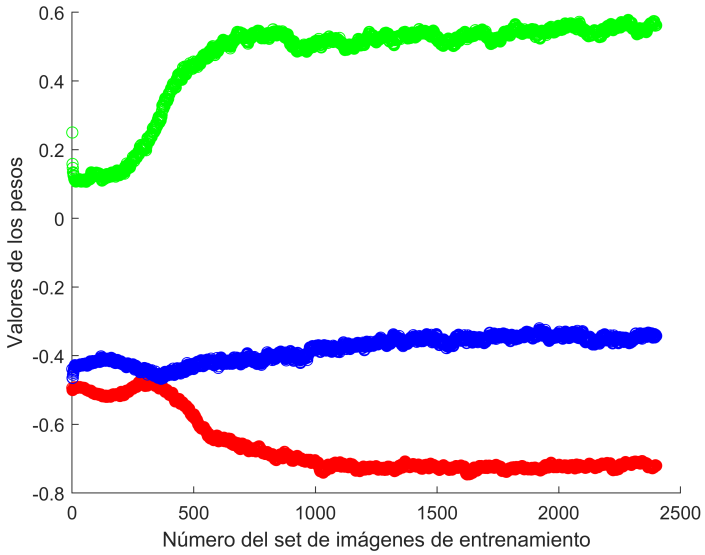


Figura 28: Evolución de los pesos durante el periodo de aprendizaje

Podemos apreciar que los valores iniciales después de un periodo inicial, sobre las 1000 iteraciones, se estabilizan alrededor de su valor final. Esta tendencia sugiere un buen aprendizaje de la red neuronal.

El código usado en este trabajo, con el que se ha llegado a estos resultados, esta disponible en el repositorio de GitHub.[56]

[56]Russell, *Código de la red neuronal*.

4. Conclusión

Por lo que se refiere a los objetivos iniciales de este trabajo, podriamos decir que se han cumplido todos. Hemos visto cómo funciona a fondo una red neuronal, profundizando en las matemáticas que la describen. Para asentar los conocimientos, hemos sido capaces de implementar con todo detalle una red neuronal que detecta digitos escritos a mano, que funciona con una precisión del 92 %.

En el campo profesional, todo el desarrollo de redes neuronales suele implementarse mediante librerías como TensorFlow, que realizan gran parte del cálculo de forma autonoma. No obstante, creo que es importante saber cómo funcionan los fundamentos de estas nuevas tecnologías, y desarrollando una red neuronal desde cero en este trabajo, satisfacemos esta necesidad completamente. Aunque la inteligencia artificial empieza a formar parte de nuestro día a día, pienso que es importante saber que el aprendizaje de las redes neuronales no es más que un programa que hace derivadas parciales. Las hará más o menos complejas, siguiendo una estructura u otra, pero en el fondo, desde un punto de vista matemático, el aprendizaje de la inteligencia artificial se basa en encontrar el mínimo de una función de coste.

En este punto nos aparece la cuestión de si la inteligéncia humana puede considerarse o no también como una forma de buscar mínimo en funciones complejas. ¿Cuándo nosotros reconocemos un número manuscrito hemos aprendido de la misma forma que hemos descrito en este proyecto? ¿Podrá la inteligencia artificial superar la inteligencia humana?

En un ambiente laboral, usar librerías como TensorFlow tiene mucho sentido, porque este código, escrito por ingenieros de Google, será mil veces mejor, más eficiente y más escalable, que el código de este trabajo. Y porque en este trabajo solo hemos estudiado la punta del iceberg. Si buscamos expandirnos a inteligencias artificiales más complejas, estas librerías se convierten en una necesidad.

En este trabajo, hemos visto la inteligencia artificial en una de sus formas más básicas posibles. Sabiendo los pasos necesarios para crear una red neuronal, se puede intuir más o menos, como se expandiría, por ejemplo, a un coche que se conduzca solo, que ha sido entrenado con diferentes situaciones, y se le ha mostrado cómo actuar. Por otro lado, tecnologías como ChatGPT se quedan lejos del alcance de este trabajo, que no resulta nada evidente como se realiza ese salto, para conseguir conversar con un programa de una forma tan convincente. Pero aun así, podemos asumir con cierta seguridad que su forma de aprender, no es más que encontrar el mínimo matemático de funciones mucho más complejas.

Tras sumergirme en este trabajo, he llegado a la conclusión de lo mucho que hay por aprender en el vasto campo de la inteligencia artificial. Por esta razón, tras realizar este trabajo me doy cuenta de que, parafraseando a Sócrates, solo sé que no sé nada.

5. Bibliografía

Referencias

History of artificial intelligence. Jun. de 2023. URL: https://en.wikipedia.org/wiki/History_of_artificial_intelligence.

Historia de la inteligencia artificial. Jun. de 2023. URL: https://es.wikipedia.org/wiki/Historia_de_la_inteligencia_artificial.

Kavlakoglu, Eda. *AI vs. Machine Learning vs. Deep Learning vs. neural networks: What's the difference?* Mayo de 2020. URL: https://www.ibm.com/cloud/blog/ai-vs-machine-learning-vs-deep-learning-vs-neural-networks.

Artificial neural network. Jun. de 2023. URL: https://en.wikipedia.org/wiki/Artificial_neural_network.

Sanderson, Grant. *Neural Networks*. Oct. de 2017. URL: https://youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi.

Wood, Thomas. *Convolutional Neural Network*. Mayo de 2019. URL: https://deepai.org/machine-learning-glossary-and-terms/convolutional-neural-network.

What are recurrent neural networks? URL: https://www.ibm.com/topics/recurrent-neural-networks.

Activation function. Mayo de 2023. URL: https://en.wikipedia.org/wiki/Activation_function.

Academy, Bot. *Neural Networks explained from scratch using Python*. Ene. de 2021. URL: https://www.youtube.com/watch?v=9RN2Wr8xvro&t=853s.

Lague, Sebastian. *How to create a neural network (and train it to identify doodles)*. Ago. de 2022. URL: https://www.youtube.com/watch?v=hfMk-kjRv4c.

Gradient. Jun. de 2023. URL: https://en.wikipedia.org/wiki/Gradient.

Training, validation, and test data sets. Jul. de 2023. URL: https://en.wikipedia.org/wiki/Training,_validation,_and_test_data_sets.

Backpropagation. Jul. de 2023. URL: https://en.wikipedia.org/wiki/Backpropagation.

Chail rule. Jun. de 2023. URL: https://en.wikipedia.org/wiki/Chain_rule.

Nayar, Shree K. *Backpropagation algorithm*. Jun. de 2021. URL: https://www.youtube.com/watch?v=sIX_9n-1UbM.

6. Modules. URL: https://docs.python.org/3/tutorial/modules.html.

TensorFlow about page. URL: https://www.tensorflow.org/about.

AssemblyAI. *How to create your first neural network with tensorflow!* Feb. de 2022. URL: https://www.youtube.com/shorts/aUlYe0Rka1o.

Choudhury, Saheli Roy. *Building an A.I. program is easy. the hard part comes after*. Jun. de 2018. URL: https://www.cnbc.com/2018/06/08/building-an-ai-program-is-easy-the-hard-part-comes-after.html.

Ráez, Arturo Montejo y Jiménez Zafra Salud María. "17. Clases y objetos". En: *Curso de Programación Python: 2019*. Anaya Multimedia, 2019.

What is overfitting? URL: https://www.ibm.com/topics/overfitting.

Russell. *Código de la red neuronal*. Sep. de 2023. URL: https://github.com/davoriols/NeuralNetworkTR.

Numpy Homepage. URL: https://numpy.org/.

numpy.dot - NumPy v1.26 Manual. URL: https://numpy.org/doc/stable/reference/generated/numpy.dot.html#numpy.dot.

What exactly does numpy.exp() do. Dic. de 2015. URL: https://stackoverflow.com/questions/31951980/what-exactly-does-numpy-exp-do#31952102.

MNIST Database python package. URL: https://pypi.org/project/mnist/.

numpy.reshape - NumPy v1.26 Manual. URL: https://numpy.org/doc/stable/reference/generated/numpy.reshape.html.

Matplotlib Homepage. URL: https://matplotlib.org/.

Stochastic gradient descent. Ago. de 2023. URL: https://en.wikipedia.org/wiki/Stochastic_gradient_descent.

numpy.genfromtxt - NumPy v1.26 Manual. URL: https://numpy.org/doc/stable/reference/generated/numpy.genfromtxt.html.

6. Anexo

El código usado en este trabajo esta disponible tanto en el repositorio de GitHub[57] como en este anexo:

6.1. Layer.py

```python
import numpy as np


class Layer:
    # generates the weights and biases of the layer
    def __init__(self, inputSize=784, outputSize=16):
        self.weights = np.random.uniform(-0.5, 0.5, (outputSize,
        ↪  inputSize))
        self.biases = np.random.uniform(-0.5, 0.5,(outputSize))
        self.outputSize = outputSize
        self.inputSize = inputSize

    # Sigmoid function to be used during the feed forward
    def sigmoid(self, x):
        return 1.0 / (1.0 + np.exp(-x))

    def feedLayerForward(self, input):
        self.activations = self.sigmoid(
            np.dot(self.weights, input) + self.biases)

        return self.activations
```

6.2. Network.py

```python
import numpy as np
from Layer import Layer


class Network:
    def __init__(self, trainInput, trainLabels, learningRate=0.01):
        # network with four layers of which the first layer is just
        ↪  the input
        # therefore we don't declare it since it doesn't have weights
        ↪  or biases
        self.layers = [Layer(784, 20), Layer(20, 20), Layer(20, 10)]
        self.trainInput = trainInput
```

[57] Russell, *Código de la red neuronal*.

```
        self.trainLabels = trainLabels
        self.learningRate = learningRate

    # Function to feedForward the network
    # Uses the output of feedLayerForward() as the input for the next layer
    def feedForward(self, input):
        self.layers[2].feedLayerForward(
            self.layers[1].feedLayerForward(
                self.layers[0].feedLayerForward(input)
            )
        )

        return self.layers[2].activations

    # function to calculate the cost of the network, not necessary for training, but
    # used to measure it
    def cost(self, image):
        self.costValue = 0

        # Calculate the cost by looping through each activation
        for activationIndex, activation in enumerate(self.layers[-1].activations):
            self.costValue += np.power(
                activation - self.trainLabels[image[0]][image[1]][
                    activationIndex
                ], 2
            )

        return self.costValue

    # function to calculate the deltas of the network ahead of time to be used in the
    # calculateGradient function
    def calculateDeltas(self, image):
        # empty list of lists, to which the deltas will be appended to the list of its
        # corresponding layer
        # \delta^2_3 will be in deltas[2][3]
        deltas = [[], [], []]

        for layerIndex, layer in zip([2, 1, 0], self.layers):
            # calculates the deltas of the last layer as they are calculated differently
            if layerIndex == 2:
                for activationIndex, activation in enumerate(
                    self.layers[-1].activations
```

```
                ):
                    deltas[2].append(
                        activation
                        * (1 - activation)
                        * 2
                        * (
                            activation
                            - self.trainLabels[image[0]][image[1]][
                                activationIndex
                            ]
                        )
                    )

            # calculates the deltas for the rest of the layers
            else:
                for activationIndex, activation in enumerate(
                    self.layers[layerIndex].activations
                ):
                    delta = 0
                    for neuronIndex, neuron in enumerate(
                        self.layers[layerIndex + 1].activations
                    ):
                        delta += (
                            deltas[layerIndex + 1][neuronIndex]
                            * self.layers[layerIndex +
                            ↪  1].weights[neuronIndex][
                                activationIndex
                            ]
                            * activation
                            * (1 - activation)
                        )
                    deltas[layerIndex].append(delta)
        return deltas

    # function to calculate the gradient vector for all the weights
    ↪  and biases
    # return a weightGradient and a biasGradient list containing the
    ↪  a multiple
    # of how much these values have to be changed in the
    ↪  backpropagation function
    #
    # therefore, the gradient vector is divided into two, to
    ↪  facilitate looping through
    # weights and biases separately
    def calculateGradient(self, image):
        # generate an empty array whose values will be changed to the
        # gradient value of the weight
        weightGradient = [[], [], []]
```

```
96
97          biasGradient = [[], [], []]
98
99          deltas = self.calculateDeltas(image)
100
101         # calculate the amount we have to change the weights:
102         for layerIndex, layer in enumerate(self.layers):
103             # checks if it is layer 0 because it need the input as an
                ↪  activation
104             if layerIndex == 0:
105                 # we calculate the gradient value for a weight
                    ↪  w_{k,j} following
106                 # a^{L-1}_j * \delta^L_k
107                 # we use trainInput as the activation from the
                    ↪  previous layer as
108                 # this is not stored as an activation
109
110                 # we reshape the information to be able to do the dot
                    ↪  product, saving a
111                 # loop
112
113                 imageShaped =
                    ↪  self.trainInput[image[0]][image[1]].reshape(1,
                    ↪  784)
114                 deltasShaped = np.array(deltas[0]).reshape(
115                     np.array(deltas[0]).shape[0], -1
116                 )
117
118                 # we do the dot product to calculate the gradient
119                 weightGradient[0] = self.learningRate * np.dot(
120                     deltasShaped, imageShaped
121                 )
122
123             else:
124                 # create variables that store the information in the
                    ↪  correct shape to do
125                 # the dot product
126                 activationsShaped = self.layers[layerIndex -
                    ↪  1].activations
127                 activationsShaped = activationsShaped.reshape(
128                     1, activationsShaped.shape[0]
129                 )
130                 deltasShaped = np.array(deltas[layerIndex]).reshape(
131                     np.array(deltas[layerIndex]).shape[0], -1
132                 )
133
134                 # we do the dot product to calculate the gradient
```

```
                weightGradient[layerIndex] = self.learningRate *
                ↪ np.dot(
                    deltasShaped, activationsShaped
                )

            # we do the same for the biases, but since there isn't
            ↪ any operations to do,
            # we can simply asign the deltas

            biasGradient[layerIndex] = self.learningRate *
            ↪ np.array(deltas[layerIndex])

        return np.array(weightGradient, dtype=object), np.array(
            biasGradient, dtype=object
        )

    # function to modify the weights and biases by a multiple of the
    ↪ gradient
    # this gradient is in theory a average of all the gradients of
    ↪ the mini set
    def backpropagation(self, weightGradient, biasGradient):
        # loop through the weights and subtract a multiple of the
        ↪ gradient value for
        # each weight
        # this can probably be done without loops, but this works
        for layerIndex, layer in enumerate(self.layers):
            for rowIndex, row in enumerate(layer.weights):
                for weightIndex, weight in enumerate(row):
                    self.layers[layerIndex].weights[rowIndex][
                        weightIndex
                    ] -= weightGradient[layerIndex][rowIndex][
                        weightIndex
                    ]

            # loop through the weights and subtract a multiple of the
            ↪ gradient value for
            # each bias
            for biasIndex, bias in enumerate(layer.biases):
                self.layers[layerIndex].biases[biasIndex] -=
                ↪ biasGradient[layerIndex][
                    biasIndex
                ]
```

6.3. main.py

```
import numpy as np
import mnist
import matplotlib
import matplotlib.pyplot as pyplot
from Network import Network

# parse the train images from the mnist database
trainImages = np.array(mnist.train_images())
# normalize the pixel values to pass them as activations
trainImages = (trainImages) / 255
# reshape the array to have 600 mini sets of 100 images
trainImages = np.reshape(trainImages, (600, 100, 784))


# parse the test images from the mnist database
testImages = np.array(mnist.test_images())
# normalize the pixel values to pass them as activations
testImages = (testImages) / 255
# reshape the array to have an array of 10000 images
testImages = np.reshape(testImages, (10000, 784))


# parse the labels form the mnist database
labels = np.array(mnist.train_labels())
# create trainLabel list with lists of zeros with the label index
↪  being 1
# for label 5, trainLabel would be [0, 0, 0, 0, 0, 1, 0, 0, 0, 0]
# do this for every label
trainLabels = []
for label in labels:
    labelList = np.zeros(10)
    labelList[label] = 1
    trainLabels.append(labelList)
# reshape labels array to have 600 mini sets of 100 labels
trainLabels = np.reshape(trainLabels, (600, 100, 10))

# get the test labels
testLabels = mnist.test_labels()

# generate network object
network = Network(trainImages, trainLabels)

# iterator to know how far is the training process
iteration = 0

for epoch in range(4):
```

```
    # Wrap everything in a loop that loops through all the sets
    for minisetIndex, miniset in enumerate(trainImages):
        iteration += 1

        # first iteration of the training
        # done outside the main loop because this way, weightGradientStack and
        # biasGradientStack isn't reasigned for each iteration, I don't know how to do
        # this in another way
        network.feedForward(trainImages[0][0])
        print(f"{iteration} --> {network.cost((0, 0))}")
        print(network.layers[2].activations)

        # I create a gradientStack variable to hold the gradient values for the weights
        # and biases
        # this is done by assigning the variable to weightGradient - weightGradientStack
        # to get an array with the same shape, where each value is 0, and doesn't affect
        # the result
        weightGradient, biasGradient = network.calculateGradient((0, 0))

        weightGradientStack = weightGradient - weightGradient
        biasGradientStack = biasGradient - biasGradient

        # loops throuhg all the images in the miniset, and calculates the weightGradient
        # biasGradient, and add them to their stack
        for imageIndex, image in enumerate(miniset):
            network.feedForward(image)

            weightGradient, biasGradient = network.calculateGradient(
                (minisetIndex, imageIndex)
            )

            weightGradientStack += weightGradient
            biasGradientStack += biasGradient

        # once all images in the miniset are trained on, we get the stack, and use
        # these values for the backpropagation function

        network.backpropagation(weightGradientStack, biasGradientStack)

```

```
# export the weights and biases to their own csv, so that the network doesn't have to be
# trained every time
np.savetxt("data/weights/weights0.csv", network.layers[0].weights, delimiter=",")
np.savetxt("data/weights/weights1.csv", network.layers[1].weights, delimiter=",")
np.savetxt("data/weights/weights2.csv", network.layers[2].weights, delimiter=",")

np.savetxt("data/biases/biases0.csv", network.layers[0].biases, delimiter=",")
np.savetxt("data/biases/biases1.csv", network.layers[1].biases, delimiter=",")
np.savetxt("data/biases/biases2.csv", network.layers[2].biases, delimiter=",")


# test the network, with the test data, that has not been seen by the network before
while True:
    imageChosen = int(input("choose a test image (0-9999)"))

    # feed forward to get activations
    network.feedForward(testImages[imageChosen])
    print(network.layers[2].activations)
    activations = list(network.layers[2].activations)

    # represent images in matplotlib:
    # reshape the image to a (28, 28) dimension array
    image = np.reshape(testImages[imageChosen], (28, 28))
    # show the image in matplotlib
    pyplot.title(
        f"label says: {testLabels[imageChosen]} \n network says: {activations.index(max(activations))}"
    )
    pyplot.imshow(image, interpolation="nearest", cmap="gray")
    pyplot.show()
```

6.4. test.py

```
import numpy as np
from numpy import genfromtxt
import mnist
from Network import Network
import matplotlib.pyplot as pyplot

# Configuration options:

# change to True to use saved training data, or False to use data from previous training
useTrainData = False

# change to True to show a pyplot of the incorrect guesses, or False to not show
showResults = False

# parse the test images from the mnist database
testImages = np.array(mnist.test_images())
# normalize the pixel values to pass them as activations
testImages = (testImages - np.min(testImages)) / (
    np.max(testImages) - np.min(testImages)
)
# reshape the array to have a list of 10000 images
testImages = np.reshape(testImages, (10000, 784))

# import test labels
testLabels = mnist.test_labels()

# generate network object
# for the methods we use to test, we don't accually need trainImages or trainLabels, but the network
# requires it, so we just pass None
network = Network(None, None)

# checks which weights and biases values we want to use
if useTrainData:
    # get the pretrained weights
    network.layers[0].weights = genfromtxt(
        "trainedData/weights/weights0.csv", delimiter=",", dtype=None
    )

    network.layers[1].weights = genfromtxt(
        "trainedData/weights/weights1.csv", delimiter=",", dtype=None
    )

    network.layers[2].weights = genfromtxt(
```

```
        "trainedData/weights/weights2.csv", delimiter=",", dtype=None
    )

    # do the same for biases
    network.layers[0].biases = genfromtxt(
        "trainedData/biases/biases0.csv", delimiter=",", dtype=None
    )

    network.layers[1].biases = genfromtxt(
        "trainedData/biases/biases1.csv", delimiter=",", dtype=None
    )

    network.layers[2].biases = genfromtxt(
        "trainedData/biases/biases2.csv", delimiter=",", dtype=None
    )

else:
    # get the user-trained weights
    network.layers[0].weights = genfromtxt(
        "data/weights/weights0.csv", delimiter=",", dtype=None
    )

    network.layers[1].weights = genfromtxt(
        "data/weights/weights1.csv", delimiter=",", dtype=None
    )

    network.layers[2].weights = genfromtxt(
        "data/weights/weights2.csv", delimiter=",", dtype=None
    )

    # do the same for biases
    network.layers[0].biases = genfromtxt(
        "data/biases/biases0.csv", delimiter=",", dtype=None
    )

    network.layers[1].biases = genfromtxt(
        "data/biases/biases1.csv", delimiter=",", dtype=None
    )

    network.layers[2].biases = genfromtxt(
        "data/biases/biases2.csv", delimiter=",", dtype=None
    )


totalGuesses = 0
correctGuesses = 0


```

```
for imageIndex, image in enumerate(testImages):
    totalGuesses += 1

    network.feedForward(image)

    activations = list(network.layers[2].activations)

    if activations.index(max(activations)) == testLabels[imageIndex]:
        correctGuesses += 1

    else:
        if showResults:
            image = np.reshape(image, (28, 28))
            # show the image in matplotlib
            pyplot.title(
                f"label says: {testLabels[imageIndex]} \n network says: {activations.index(max(activations))}"
            )
            pyplot.imshow(image, interpolation="nearest", cmap="gray")
            pyplot.show()

    print(f"correct guesses: {correctGuesses}")
    print(f"total guessses:  {totalGuesses}")
    print(f"accuracy:        {correctGuesses / totalGuesses * 100}")


# test the network, with the test data, that has not been seen by the network before
while True:

    imageChosen = int(input("choose a test image (0-9999)"))

    # feed forward to get activations
    network.feedForward(testImages[imageChosen])
    print(network.layers[2].activations)
    activations = list(network.layers[2].activations)

    # represent images in matplotlib:
    # reshape the image to a (28, 28) shape array
    image = np.reshape(testImages[imageChosen], (28, 28))
    # show the image in matplotlib
    pyplot.title(f"label says: {testLabels[imageChosen]} \n network says: {activations.index(max(activations))}")
    pyplot.imshow(image, interpolation="nearest", cmap="gray")
    pyplot.show()
```