CREACIÓN DE UNA RED NEURONAL EN PYTHON PARA IDENTIFICAR NÚMEROS ESCRITOS A MANO

Adaptación del capítulo 1 del libro de Michael Nielsen

ÍNDICE

INTRODUCCIÓN

El sistema de visión del ojo humano es una de las grandes maravillas del mundo, consideremos la siguiente secuencia de dígitos:

La gran mayoría de gente es capaz de reconocer estos dígitos como 504192, algo que nos parece trivial pero técnicamente no lo es tanto. En cada hemisferio de nuestro cerebro, las personas tenemos un córtex visual primario con 140 millones de neuronas, que a su vez tienen decenas de billones de conexiones entre ellas, tenemos en nuestras cabezas un super ordenador, el cual se ha ido modificando y mejorando durante cientos de millones de años, y se ha ido adaptando al mundo visual.
Por eso mismo, las personas somos muy buenos para dar significado a lo que nuestros ojos nos suelen mostrar, y casi todo el trabajo se hace de manera inconsciente, y es por eso que no apreciamos realmente como de complicados son los problemas que nuestro sistema visual resuelve.

Esta dificultad se hace aparente si se trata de escribir un programa que se encarga de reconocer dígitos escritos a mano, y es que lo que parecía sencillo para nosotros mismos, de repente se convierte en algo extremadamente complicado, ya que simples intuiciones como "un 9 tiene un circulo arriba y un palo vertical abajo" no son tan "simples" de expresar dentro de un algoritmo, y al tratar de hacer dichas reglas precisas, empiezan a surgir excepciones y cosas a tener en cuenta, haciendo la tarea prácticamente desesperanzadora.

En cambio, las redes neuronales observan el problema desde otro punto de vista. La idea que tienen estas redes neuronales es tomar una gran cantidad de dígitos escritos a mano (conocidos como ejemplos de entrenamiento), y a partir de ellos desarrollar un sistema en el que se pueda aprender a partir de dichos ejemplos de entrenamiento.
En otras palabras, las redes neuronales utilizan los ejemplos que se les proporcionan para deducir reglas para reconocer dichos dígitos. Además, si se aumenta la cantidad de ejemplos de entrenamiento, la red puede aprender más sobre la escritura a mano, y por tanto incrementar su porcentaje de acierto.

El objetivo de este trabajo va a ser escribir un programa de Python que implemente una red neuronal que aprenda a reconocer dígitos escritos a mano. Se trata de un programa no muy complejo y para nada largo, pero que acaba teniendo un porcentaje de deducción del **96%** (pudiendo incrementar dicho porcentaje hasta el **99%** si se hicieran futuras mejoras, aunque no se contemplarán porque es ya mucho trabajo y esfuerzo para lo "poco" que se gana).
Este tipo de redes neuronales están siendo usadas ya hoy en día en el mundo, como por ejemplo en bancos para procesar cheques y en oficinas de correos para reconocer direcciones postales.

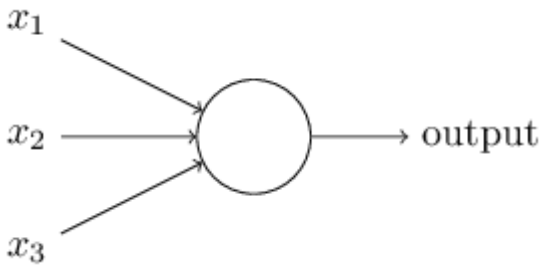
La razón de porqué decidir hacerlo sobre escritura a mano en vez de sobre números digitales es porque, supone más reto que sobre números digitales pero al mismo tiempo no tiene una extremada dificultad como para que resulte imposible de realizar.
De hecho, esta aplicación es una buena manera de desarrollar técnicas más avanzadas de aprendizaje, como el ***deep learning***, que se discutirá más adelante.

Para que este documento explicación del trabajo no quede tan corto y limitado, se van a desarrollar las ideas clave más importantes que se han tomado en consideración durante el proyecto, además de incluir una explicación sobre dos tipos importantes de neuronas artificiales (perceptrones y neuronas sigmoide), y sobre el modelo gradiente descendente estocástico.
Es por eso que el documento va a ser más largo de lo que se puede esperar, pero a cambio, el lector podrá entender al final del trabajo lo que realmente el aprendizaje ***deep learning*** es y porqué es tan importante.

PERCEPTRONES ¿QUÉ SON?

¿Qué es realmente una red neuronal? Para comenzar, se va a explicar lo que es un tipo de neurona artificial llama ***perceptrón***, que fueron creados por Frank Rosenblatt entre 1950 y 1960, pero hoy en día es más común el uso de otras neuronas artificiales, como por ejemplo la ***neurona sigmoide***, que explicaremos a continuación, pero para entenderlas, primero hay que entender los ***perceptrones***.

Entonces, ¿cómo funcionan los ***perceptrones***? Un ***perceptrón*** toma diferentes inputs (x1, x2, ...) y produce un simple output:



Rosenblatt propuso una simple regla para calcular el output de los ***perceptrones***, los pesos (***weights***), números reales que expresan la importancia de los inputs en el output total. El output de la neurona (0 o 1) se determina por la suma total de los pesos y se compara con un determinado valor para ver si es mayor o menor (***threshold***). Expresado en términos algebráicos:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

Y esta es toda la complejidad del funcionamiento de los ***perceptrones***.

Vamos a ver un ejemplo para entenderlo de una forma más clara. Imagina que te invita a ir un fin de semana al pueblo un amigo tuyo, y tienes que tener en cuenta varios factores.

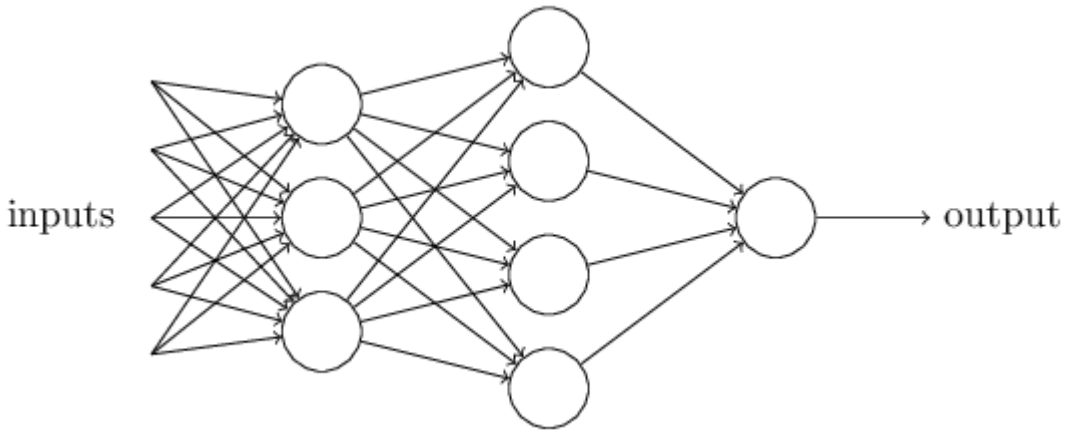
1. ¿Hará buen tiempo?
2. ¿Irá más gente o solo sereis tu amigo y tú?
3. ¿Se puede ir en transporte público hasta el pueblo?

Podemos representar estos valores por x1, x2 y x3. Dichos valores serían, por ejemplo, x1 = 1 si hará buen tiempo o x1 = 0 si hará mal tiempo. De la misma manera, x2 = 1 si vas con más amigos o x2 = 0 si vas solo. Y similarmente también para x3 si hay o no transporte público.

Ahora imagina que sabes que en el pueblo de tu amigo siempre te lo pasas bien aunque no vayan muchos amigos tuyos, pero que haga mal tiempo te fastidia el plan por completo porque no podrías ir a ningún lado, entonces es cuando puedes usar los perceptrones para tomar una decisión.
Para ello, podríamos poner que el peso w1 = 6 para si hace buen tiempo, y w2 = 2 y w3 = 2. Esto significa que el estado del tiempo es muy importante para este plan, pero que vaya más gente o la posibilidad de ir en transporte público no representa tanta importancia.
Finalmente, imagina que eliges como ***threshold*** el número 5, esto dará como resultado que todo dependa del tiempo en este caso, si x1 = 1 el plan siempre saldrá adelante y si x1 = 0 el plan no saldrá adelante.

A parte de este resultado obtenido, también podemos variar los pesos y el ***threshold*** para conseguir diferentes resultados y modos de tomar decisiones, como por ejemplo poner el ***threshold*** a 3, entonces si tus amigos van y hay transporte público (x2 = 1 y x3 = 1) el plan saldrá adelante aunque haga mal tiempo también, lo cual generará otra forma de tomar decisiones.
Así mismo, cuanto menor sea el ***threshold*** significa que más ganas tendrás de que el plan salga adelante.

Aun así, obviamente los ***perceptrones*** no es un modelo completo de pensamiento y toma de decisiones humanas, pero el ejemplo ilustra cómo un ***perceptrón*** puede dar más peso o menos a diferentes valores para tomar decisiones, y por tanto se puede deducir que una red de ***perceptrones*** más compleja dé lugar a decisiones bastante buenas y decentes:



En esta red que podemos observar aquí, la primera columna de ***perceptrones*** (la primera ***capa***) toma tres simples decisiones dando peso a los diferentes inputs, pero, ¿qué hacen los ***perceptrones*** de la segunda columna?
Bien pues, cada uno de estos ***perceptrones*** toma una decisión en función de la decisión tomada en la anterior capa, lo que resulta en que puedan tomar decisiones a un nivel más complejo y más abstracto que los ***perceptrones*** de la primera capa, y así cuantas más capas, más complejidad y más sofisticación.

Se ha mencionado antes que los ***perceptrones*** tienen un solo output y en esta red hay diferentes outputs por ***perceptrón***, pero son todos el mismo output, solo que para que visualmente se entienda mejor se representa así que el output de uno se utiliza como input en varios otros ***perceptrones***, es más claro que dibujar una sola línea que luego se divide en varias.

Vamos a simplificar ahora la manera en la que describimos a los perceptrones, la manera de calcular el output explicada anteriormente:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

Se puede modificar y expresar ese sumatorio como un producto escalar de w por x.

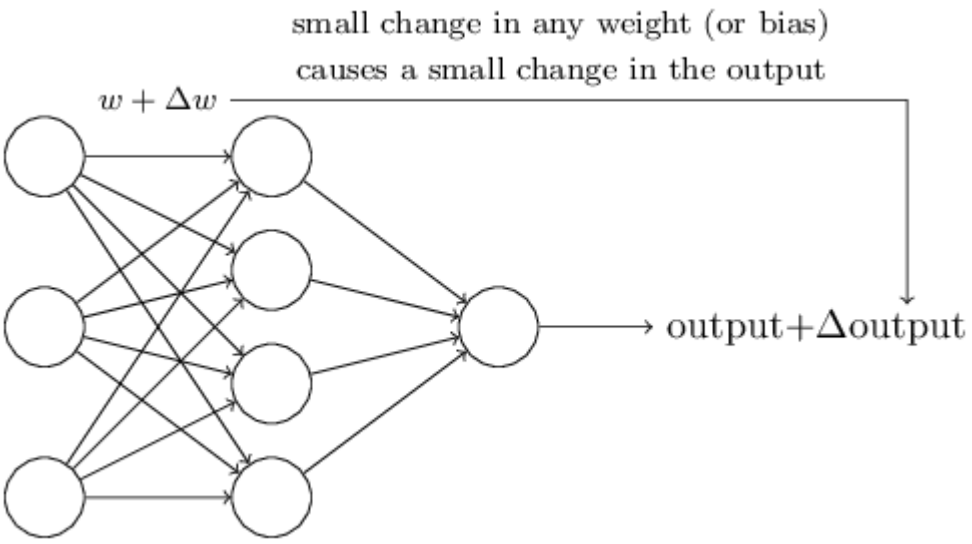
Además, el ***threshold*** se puede mover al otro lado del signo mayor/menor, y pasaría a ser la tendencia (***bias***) del ***perceptrón***, quedando la fórmula así:

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Este ***bias*** se puede tomar como cómo de fácil es que un ***perceptrón*** de 1 como output, ya que por ejemplo, para un ***perceptrón*** con un ***bias*** muy elevado, es muy fácil que dé como output 1, pero si dicho ***bias*** es muy negativo, entonces que dé como output 1 es realmente complicado.
Esta introducción de los ***bias*** es solo un cambio pequeño en los ***perceptrones***, pero veremos como afecta en el futuro (a partir de ahora se utilizará siempre ***bias*** en vez de ***threshold***).

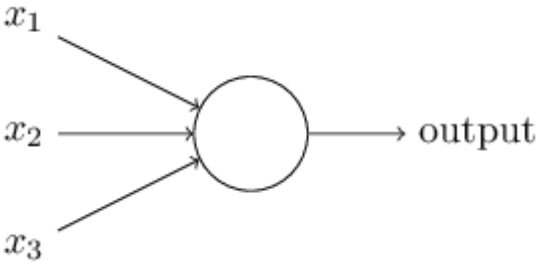
NEURONAS SIGMOIDE ¿QUÉ SON?

Vamos a imaginar que tenemos una red de ***perceptrones*** que queremos usar para solucionar un determinado problema, por ejemplo los inputs de la red pueden ser los datos de los píxeles de una imagen escaneada de un dígito escrito a mano, y queremos que la red aprenda con los pesos y los ***bias*** para que el output sea la clasificación correcta de ese mismo dígito escrito a mano.
Entonces, lo que buscamos es, ir haciendo pequeños cambios en los pesos para generar un pequeño cambio correspondiente en el output de la red, lo cual hará posible el aprendizaje de dicha red:



El problema es que, esta capacidad de producir cambios menores en el output realizando cambios menores en los pesos, no es lo que pasa con la red cuando contiene ***perceptrones***, en esta red esto causaría que el output cambiará completamente de 0 a 1 (únicos valores que puede tomar el output con los ***perceptrones***.

La manera en la que podemos solventar este problema es introducir un nuevo tipo de neuronas, las neuronas ***sigmoide***. Estas neuronas son similares a los perceptrones, pero modificadas para que puedan sufrir pequeños cambios en sus pesos y ***bias*** y originar así solamente pequeños cambios en el output, ese es el factor crucial que les hará aprender.



Las neuronas ***sigmoide*** se pueden representar igual que los ***perceptrones***, ya que para varios inputs (x1, x2,...) dará un solo output, pero en este caso los inputs pueden ser CUALQUIER valor entre 0 y 1, no sólo 0 y 1 (por ejemplo 0.638 sería un input válido).

Además, igual que los ***perceptrones***, las neuronas ***sigmoide*** tendrán un peso por cada input w1, w2,... y un ***bias*** total b, y el output que generarán no será 0 o 1, sino $\sigma(w \cdot x+b)$, y σ sería lo que llamamos la ***función sigmoide***, definida por:

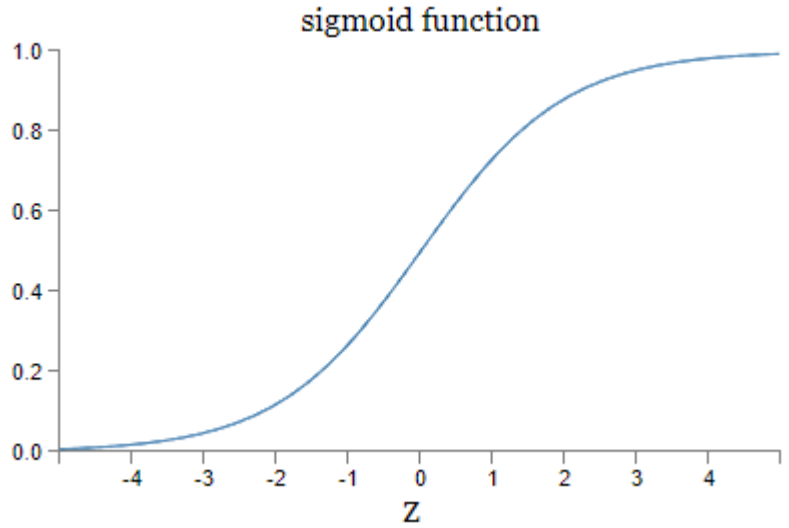
$$\sigma(z) \equiv \frac{1}{1+e^{-z}}.$$

Y el output total de una neurona ***sigmoide*** con sus inputs x1, x2,..., pesos w1, w2,... y bias b es:

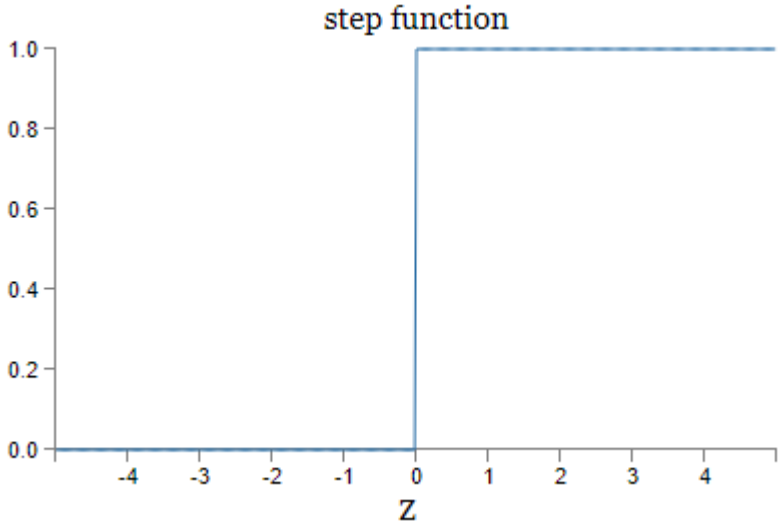
$$\frac{1}{1+\exp(-\sum_j w_j x_j - b)}.$$

A primera vista, puede parecer que las neuronas ***sigmoide*** son muy diferentes de los ***perceptrones***, y que su fórmula algebraica es opaca y difícil de entender si no estás familiarizado con ella, pero la verdad es que hay incontables similitudes entre los ***perceptrones*** y las neuronas ***sigmoide***, y que la fórmula algebraica de las neuronas ***sigmoide*** es más un detalle técnico que una barrera para el entendimiento.

La verdad es que la fórmula en sí de σ no es tan importante, lo que de verdad importa es la forma de la función que traza:



La cual es una versión "suavizada" de la función ***step***:

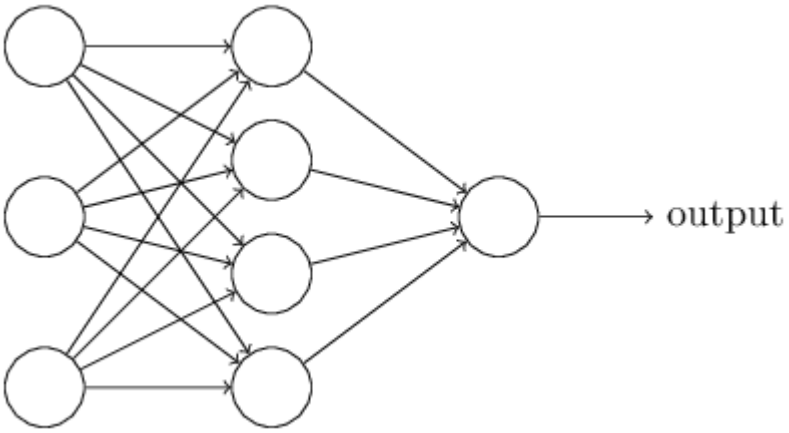


De hecho, si σ fuera la función ***step***, entonces la neurona ***sigmoide*** SERÍA un ***perceptrón***, ya que el output sería 0 o 1 dependiendo de si $w \cdot x+b$ es positivo o negativo.
El resultado que obtenemos al usar la función actual de σ es un ***perceptrón*** "suavizado", y esta "suavidad" es el factor clave, es lo que hace que pequeños cambios en los pesos de cada neurona y en los ***bias*** produzcan diferentes cambios en el output de esta misma.

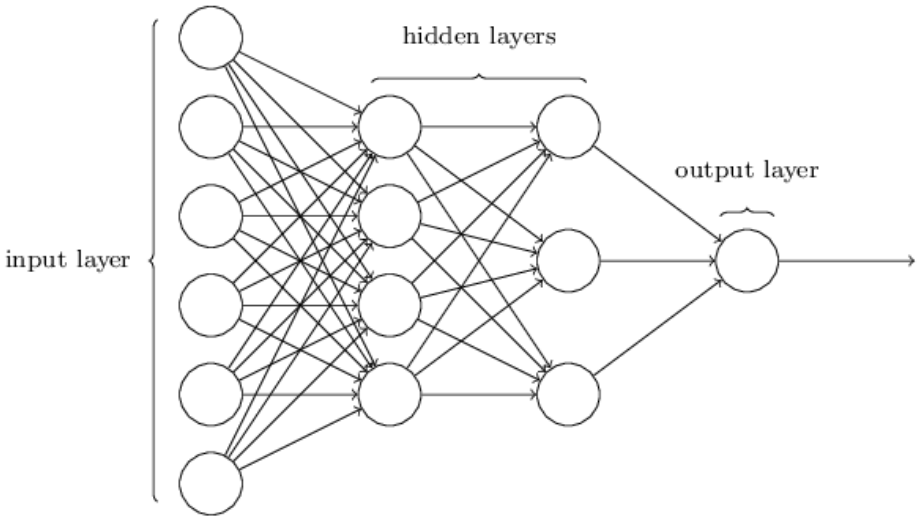
¿Y cómo se interpreta el output de una neurona ***sigmoide***? Obviamente, hay una gran diferencia entre los ***perceptrones*** y las neuronas ***sigmoide*** y es que estas últimas no solo producen un 0 o un 1, y pueden producir cualquier número real entre 0 y 1 (por ejemplo 0.173, 0.689...), lo cual resulta la pieza clave para casos en los que, por ejemplo, se quiere representar la intensidad media de los píxeles de una imagen introducida como parámetro en una red neuronal.

LA ARQUITECTURA DE LAS REDES NEURONALES

En este apartado, se va a explicar, describir y mostrar una red neuronal que se encarga de clasificar números escritos a mano, pero antes que nada, primero vamos a dejar clara cierta terminología de las redes neuronales.
Imaginemos que tenemos esta red:



Las neuronas de la izquierda, las que reciben el input, reciben el nombre de ***neuronas de entrada***, y la neurona de la derecha, la que produce el resultado, recibe el nombre de ***neurona de salida***. Las capas intermedias se llaman ***neuronas ocultas*** y no son ni de entrada ni de salida.
En este caso, la red de arriba solo tiene capa de ***neuronas ocultas***, pero eso ya depende de la complejidad de la que se quiera dotar a la red neuronal, por ejemplo esta de a continuación tiene dos capas de ***neuronas ocultas***:



La forma en la que están diseñadas las neuronas de entrada y de salida es muy simple, imaginemos que queremos determinar si una determinada imagen muestra un "9" escrito a mano, para ello la manera en la que se programará la red neuronal es codificar las intensidades de los píxeles en cada una de las neuronas de entrada.
De esta manera, si la imagen es de tamaño 64x64 píxeles, entonces habrá 4.096 neuronas de entrada (64x64 = 4.096), y cada una recibirá una intensidad escalada aproximadamente entre 0 y 1 (gracias a que son neuronas *sigmoide*).
De la misma manera, la neurona de salida será una simple neurona que, si su valor es superior a 0.5 entonces la imagen introducida será un "9", mientras que si su valor es inferior a 0.5 entonces la imagen introducida no es un "9".

Dejando ahora claro el diseño de las neuronas de entrada y de salida, vamos a pasar ahora a hablar de las *neuronas ocultas*, cuyo diseño sí que tiene un poco más de complejidad; de hecho, no es posible decidir el comportamiento de las *neuronas ocultas* con unas determinadas reglas preestablecidas.
En lugar de eso, los investigadores y expertos en redes neuronales han desarrollado muchos diseños heurísticos para *neuronas ocultas*, para que todo el que quiera pueda conseguir el comportamiento que quiere en su red neuronal.
Estas heurísticas también pueden ser utilizadas para ayudar a determinar la cantidad de *neuronas ocultas* exactas que se tienen que utilizar en una determinada red neuronal para no entorpecer el tiempo necesario para posteriormente entrenarla.

Hasta ahora se ha hablado de redes neuronales en las que la salida de una neurona es la entrada de otra de la siguiente capa; este tipo de redes neuronales se llaman redes *feedforward*, y significa que no hay bucles en la red, la información y los datos siempre avanzan hacia delante, nunca hacia atrás.

De hecho, si hubiera algún bucle por algún motivo nos podríamos encontrar en situaciones en las que el *input* de las funciones σ (sigmoide) dependiera del *output*, lo cual sería muy complicado de hacer que tuviera sentido, así que en nuestro caso es mejor que no se permita ningún tipo de bucle.

Aún así, es importante mostrar al lector que en otros tipos de modelos artificiales de redes neuronales los bucles sí que son posibles y útiles, y reciben el nombre de redes neuronales recurrentes.
La idea de estos modelos es tener neuronas que producen resultados durante una determinada cantidad de tiempo y luego se detienen y se

quedan inactivas, y en esa cantidad de tiempo que han estado activas produciendo resultados han ido activando otras neuronas, que también se activan durante un pequeño periodo de tiempo, y así sucesivamente, creando una ola de neuronas activándose y apagándose.
En estos casos, los bucles no generan ningún tipo de problema, ya que el ***output*** de una neurona solo afecta a su ***input*** en otro momento más tardío, no en el mismo instante.

Las redes neuronales recurrentes han sido menos influyentes que las ***feedforward***, en parte porque los algoritmos de aprendizaje para las redes recurrentes son (de momento) menos potentes y capaces de realizar tareas más complejas, pero aún así son muy interesantes y mucho más similares a la forma de trabajar que tiene nuestro cerebro que las ***feedforward***.
Además, también es posible que las redes neuronales recurrentes resuelvan problemas importantes que no pueden ser resueltos con las redes ***feedforward*** sino es con gran dificultad, pero bueno, el objetivo principal de este trabajo son las vastamente utilizadas redes ***feedforward***.

GENERACIÓN DE UNA SIMPLE RED NEURONAL

Tras haber definido las redes neuronales, vamos ahora a volver al tema de reconocer la escritura a mano de las personas, problema que se puede dividir en dos sub problemas.
En primer lugar, tenemos que saber dividir una imagen conteniendo diferentes dígitos en imágenes separadas de un único dígito, por ejemplo, tenemos que dividir la imagen

en seis imágenes separadas,

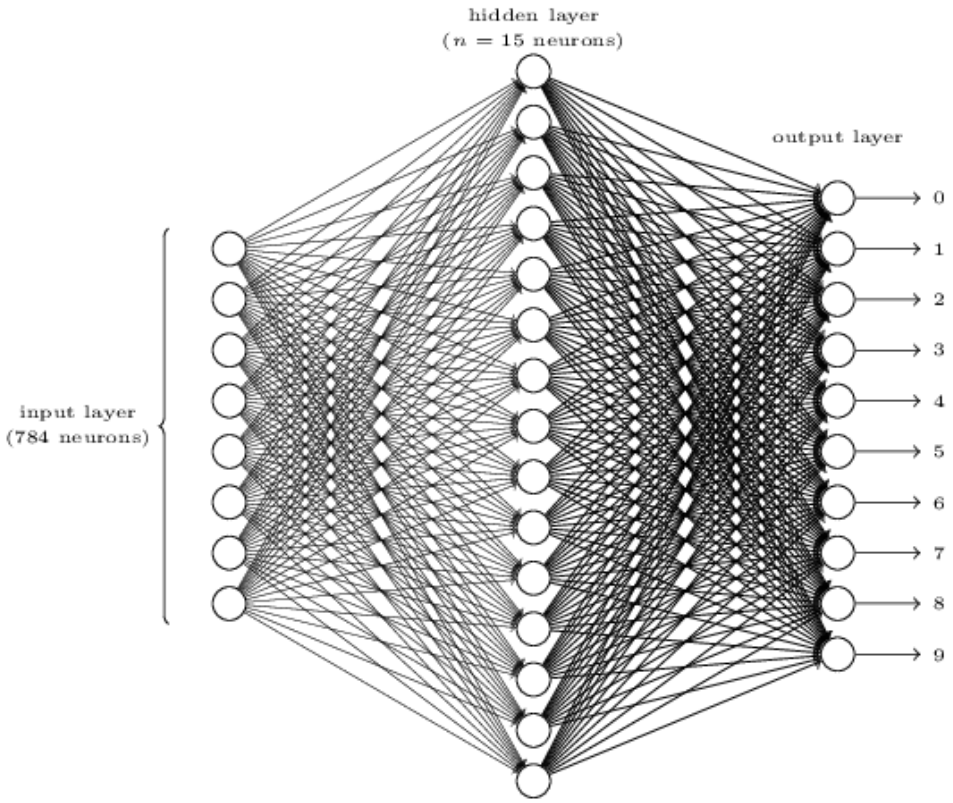
Claro que este tema, la segmentación en diferentes dígitos, es algo que nosotros los seres humanos podemos resolver sin ninguna dificultad, pero es algo muy complicado para un programa recortar correctamente una imagen.
En segundo lugar, una vez la imagen ha sido dividida, el programa tiene que saber clasificar cada número individualmente. Por ejemplo, lo idóneo sería que nuestro programa viera la siguiente imagen

y supiera identificarla como el número 5.

Vamos a centrarnos en el segundo problema, clasificar los números individualmente, ya que el problema de la segmentación pasa a ser trivial una vez tienes una buena forma de clasificar los números individualmente, debido a las muchas soluciones que podemos aplicar para realizarlo (como por ejemplo, dividir aleatoriamente y descifrar si en la zona dividida se encuentra un dígito, decidiendo así si la segmentación ha sido realizada con éxito o no), así que, en vez de

preocuparnos por la segmentación, vamos a preocuparnos por desarrollar una red neuronal que pueda averiguar un problema más complicado, el reconocimiento individual de dígitos escritos a mano.

Para conseguir reconocer dígitos individuales, vamos a hacer uso de una red neuronal que tiene tres capas:



La capa de entrada (*input layer*) contiene las neuronas codificando el valor de los píxeles de la imagen introducida en ese momento, que en este caso sería una imagen de 28x28 píxeles, así que habría 784 neuronas de entrada (28x28 = 784) cada una de ellas con un valor determinado entre 0 y 1 en función del valor *grayscale* de cada píxel (0.0 representa el blanco y 1.0 representa el negro).

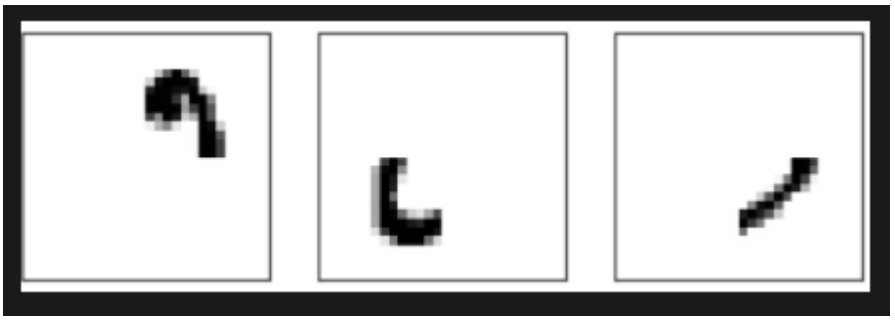
La segunda capa es la capa oculta (*hidden layer*), y la cantidad de neuronas en esta capa viene denotada por *n*, y se podrá experimentar con diferentes valores de *n* (en la imagen por ejemplo, toma el valor de *n* = 15).

Finalmente, la capa de salida (*output layer*) contiene 10 neuronas, una por cada número del 0 al 9, y dependiendo de cual acabe con el mayor resultado el número correspondiente a esa neurona será el que la red neuronal habrá identificado con la imagen introducida.

Pero el lector se puede preguntar, ¿por qué 10 neuronas de salida? Si el objetivo es decir que número ha sido identificado (0, 1, 2,..., 9) se podrían utilizar 4 neuronas de salida, tratando cada neurona como si fuera un valor binario dependiendo si el output está más cercano a 0 o a 1.
Entonces, ¿por qué nuestra red tiene que utilizar 10 neuronas de salida? ¿No sería esto ineficiente? Pues bien, la justificación es empírica: se pueden probar las dos formas y el resultado final es que la red con 10 neuronas de salida funciona mucho mejor que la que tiene 4, ¿hay alguna heurística que nos diga con antelación porque deberíamos usar 10 neuronas de salida en vez de 4?

Pues para entender esto, va a ayudar pensar sobre cómo trabaja la red neuronal desde su base más simple, vamos a poner que esta vez tenemos 10 neuronas de salida.
Bien pues, en primer lugar vamos a concentrarnos en la primera neurona de salida, que es la encargada de decidir si el dígito es o no es un 0. La manera en la que realiza esto es sopesando los valores que le han sido pasados desde la capa ***oculta*** de neuronas, ¿y qué están haciendo las neuronas de dicha capa ***oculta***? Bien pues, supongamos que la primera neurona ***oculta*** de la capa ***oculta*** detecta si la imagen tiene un segmento como el siguiente:

Para ello, podemos darle un mayor valor a los pesos de los píxeles que intersectan con los de la imagen en negro, y dandole menor valor al resto de pesos de los otros inputs.
De manera semejante, digamos que la segunda, tercera y cuarta neuronas ***ocultas*** detectan si los siguientes segmentos están presentes:



Como habrás podido deducir, las cuatro imágenes juntas forman un 0

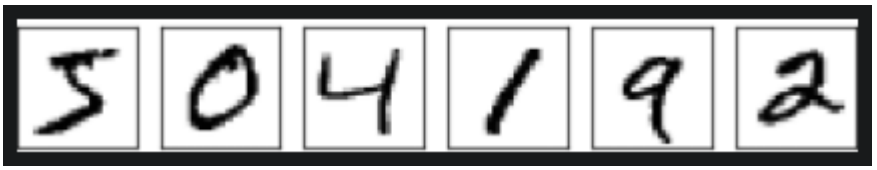
Así que si estas cuatro neuronas ***ocultas*** mencionadas producen el mismo resultado positivo a la vez, podemos concluir que el dígito en este caso es un 0.
Por supuesto, esta no es la ÚNICA prueba para concluir que esta imagen presenta un 0, se puede escribir un 0 de muchas otras maneras (moviéndolo un poco de posición, deformando ligeramente el número), pero es seguro asumir que en este caso el número introducido es un 0.

Suponiendo que la red neuronal funcione así, se puede dar una explicación plausible de porque es mejor tener 10 neuronas de salida en vez de 4, ya que si tuviéramos 4, la primera neurona se encargaría de decidir cuál es el bit más significativo del dígito, y no hay manera fácil de relacionar su bit más significativo con la forma que presenta el dígito.

Ahora, con todo dicho, esto es todo simplemente heurístico, nada nos confirma que una red neuronal de tres capas funcione de la manera en la que se ha explicado en este documento, con las neuronas ***ocultas*** detectando simples formas y figuras parciales de un dígito.
Quizá un algoritmo de aprendizaje inteligente puede encontrar la manera de asignar los pesos para una red neuronal en la que sólo hay 4 neuronas de salida, pero de manera heurística, la manera explicada para hacer funcionar la red neuronal funciona de forma bastante correcta, y se puede ahorrar una gran cantidad de tiempo diseñando una buena arquitectura para la red neuronal.

APRENDIENDO CON GRADIENTE DESCENDENTE (*GRADIENT DESCENT*)

Ahora que hemos diseñado la red neuronal, ¿cómo podemos hacer que reconozca dígitos?
La primera cosa que vamos a necesitar es un determinado ***dataset*** para que la red neuronal vaya aprendiendo de algún lugar, y para ello vamos en este proyecto se va a utilizar el MNIST dataset, que contiene decenas de miles de imágenes escaneadas de dígitos escritos a mano y su correcta clasificación (***mnist.pkl.gz***).
El nombre de MNIST viene del hecho de que se trata de un set de datos proporcionado y coleccionado por NIST, el Instituto Nacional de Estándares y Tecnología de los Estados Unidos, aquí hay algunas imágenes de ejemplo:



Estos son, efectivamente, los dígitos que se han ido mostrando a lo largo de todo el documento, y serán los que utilizará nuestro programa para entrenar.

Este ***dataset*** del MNIST está dividido en dos partes, la primera contiene 60.000 imágenes que van a ser usadas para entrenar a la red neuronal, son imágenes escaneadas de dígitos escritos a mano por 250 personas, tanto de adultos como de estudiantes de instituto, y son imagenes en blanco y negro de 28 por 28 píxeles.
La segunda parte son 10.000 imágenes que van a ser usadas como imágenes de prueba, y van a servir para evaluar cómo de bien la red neuronal ha aprendido a reconocer dígitos.
Para que sea más eficaz, los dígitos de esta segunda parte han sido extraídos de otras 250 personas DIFERENTES a las 250 elegidas para hacer las imágenes de entrenamiento, hecho que nos dará más confianza de que nuestro sistema podrá incluso reconocer dígitos de gente cuya escritura no ha visto durante el entrenamiento.

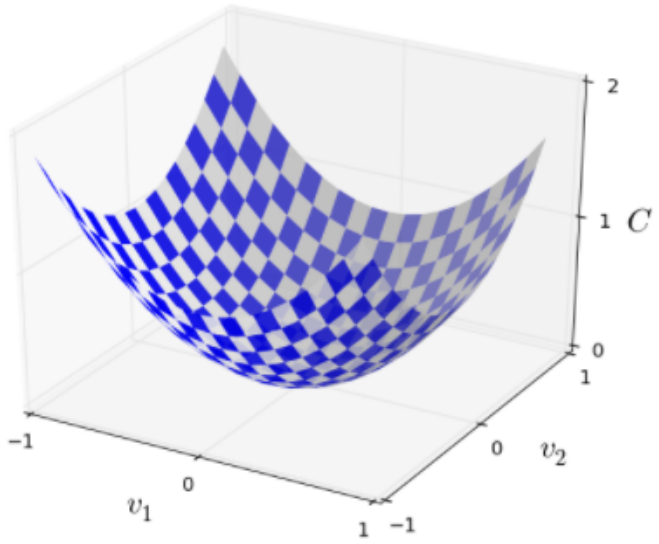
Entonces ahora, lo que queremos es un algoritmo que nos permita descifrar que ***weights*** y ***biases*** necesitamos para que el resultado de salida de la red neuronal sea el correcto para todo dato que se le introduzca.

Para cuantificar cómo de bien se va a conseguir esto, se define la siguiente función de *coste*:

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2.$$

Aquí, la *w* es la colección de todos los pesos de la red, *b* la colección de todos los *biases*, *n* es la cantidad total de datos de entrenamiento introducidos, *a* es el vector de las neuronas de salida para un determinado input *x* (por ejemplo, si introducimos el dígito 5 como input *x*, entonces *a* debería ser igual a (0, 0, 0, 0, 0, 1, 0, 0, 0, 0)) y el sumatorio es la cantidad total de datos de entrada introducidos.
A *C* lo llamaremos la función de coste *cuadrático*, e inspeccionando dicha fórmula veremos que *C(w, b)* nunca será negativo, debido a que ninguno de sus componentes en el sumatorio lo es.
De hecho, *C(w, b)* tenderá a 0 justo cuando *y(x)* es aproximadamente igual al resultado de salida *a* para todos los datos de entrada introducidos, es decir, que cuanto mejor ajustados están los pesos y los *bias*, más cercana a cero dará como resultado esta función, y por contra, cuanto más valga *C*, pero lo estará haciendo el algoritmo para identificar los pesos y los *bias*.
Es decir, el objetivo principal será buscar un conjunto de pesos y de *bias* cuyo valor haga que se reduzca el coste de la función al mínimo posible, y esto lo hacemos con el *gradiente descendente*.

Pero bueno, esto era solo a modo de curiosidad de cómo introducimos el *gradiente descendente* en nuestro sistema, vamos ahora a simplificar la función para que resulte más sencillo de entender y no haya tantas variables en juego.
Vale pues, vamos a suponer que nuestro objetivo es tratar de reducir una determinada función, *C(v)* (esto puede funcionar para cualquier función de tantos valores como sea *v = v1, v2,...*) (destaco que se ha sustituido *w* y *b* por *v*, ya no estamos tanto dentro del contexto de redes neuronales, sino en un contexto más amplio, para que se entienda mejor).
Entonces, para tratar de reducir *C(v)* quizá ayudaría imaginar una función de dos variables, *v1* y *v2*:



Y el objetivo que tenemos es, cómo puede resultar obvio, tratar de que *C* llegue al mínimo global de la función.
Bueno, en este ejemplo en particular quizá es muy sencillo, pero muchas veces *C* será una función mucho más complicada y con muchas más variables (como es nuestro caso con la red neuronal), y no será tan sencillo encontrar el punto mínimo.

Una manera de afrontar este problema podría ser usar álgebra y cálculo para encontrar el mínimo analíticamente, podemos calcular las derivadas y tratar de encontrar los sitios en los que *C* es un extremo, y esto puede funcionar si, con suerte, *C* es una función con pocas variables, pero esto es un problema cuando *C* tiene una cantidad muy elevada de variables, hecho que sucede con las redes neuronales (como es nuestro caso, ya que el resultado depende de billones de pesos y ***bias*** calculados de una forma extremadamente complicada), esta opción no es eficaz en nuestro caso.

Habiendo descartado el cálculo, vamos a pensar en el problema de una manera mucho más visual.
Imaginemos, que nuestra función es como una especie de valle entre dos montañas, lo lógico es que si tu colocas una pelota en dicho valle, esta tienda a ir hacia el punto más bajo del valle, e irá bajando poco a poco por él. ¿Podemos utilizar esta idea para encontrar el mínimo de la función? Vamos a probar.
Empezaremos eligiendo un punto aleatorio para esta pelota (imaginaria), y luego simularemos como si fuera rodando hacia abajo. Para realizar esta simulación podemos calcular las derivadas (y quizá segundas derivadas) de *C*, y estas derivadas nos dirán lo que

necesitamos saber sobre la forma del valle y, por tanto, como deberá rodar nuestra pelota para llegar hasta el fondo.

Vamos a probar a ver que sucede cuando movemos la pelota un determinado incremento en dirección *v1* y otro determinado incremento en dirección *v2*, la fórmula cambiará de la siguiente manera:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2.$$

Entonces, vamos a tratar de elegir dos incrementos que hagan que *C* vaya decreciendo (para hacer que la pelota vaya rodando cuesta abajo en el valle).
Para descifrar cual es la elección que tenemos que tomar para que esto suceda, nos va a ayudar definir el incremento total de *v* de la manera siguiente *Δv ≡ (Δv1, Δv2)^T* donde la *T* es la traspuesta de la operación, que cambiará los vectores fila en vectores columna.
También vamos a definir el *gradiente* de *C* como el vector de las derivadas parciales, definiremos este vector gradiente con *∇C*:

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T.$$

Antes de continuar, voy a definir lo que significa el símbolo *∇* para que quede claro. De hecho, *∇C* se puede definir como un objeto matemático único (el vector definido en la fórmula de arriba), pero *∇* por si mismo es como una marca que te dice, "oye, *∇C* es un vector gradiente".

Combinando ahora las definiciones dadas hasta la fecha, *ΔC* se puede definir como:

$$\Delta C \approx \nabla C \cdot \Delta v.$$

Y esta ecuación muestra por qué *∇C* recibe el nombre de vector gradiente: *∇C* relaciona los cambios de *v* con cambios de *C*, que es lo que esperaríamos que hiciera un gradiente. Pero lo realmente interesante de esta ecuación es que nos deja ir variando *Δv* para ir haciendo *ΔC* cada vez más negativo.

En concreto, suponemos que elegimos

$$\Delta v = -\eta \nabla C,$$

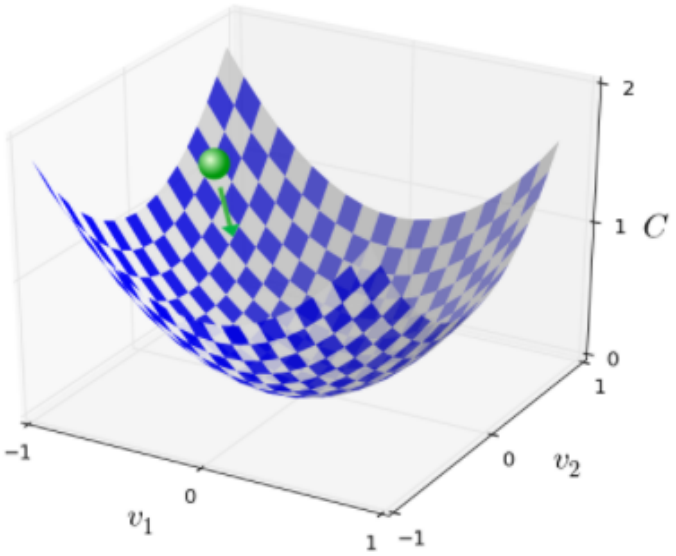
donde η es un pequeño valor positivo (conocido como *learning rate*). Entonces, la ecuación anterior ($\Delta C \approx \nabla C \cdot \Delta v$) se puede modificar por

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2$$

Y de esta manera, ya que ||∇C||^2 siempre va a ser mayor o igual que cero, esto nos asegura que ΔC siempre va a ser menor o igual que cero, es decir, que C siempre va a decrementar, nunca incrementará si variamos v como hemos definido antes, en función de η.
¡Esta es exactamente la propiedad que íbamos buscando! Esta función $\Delta v = -\eta \nabla C$ definirá la "ley de movimiento" de la pelota en nuestro algoritmo de gradiente descendente, de esta manera calcularemos un valor para Δv y luego moveremos la pelota en dirección v en esa cantidad:

$$v \to v' = v - \eta \nabla C.$$

De esta manera, iremos actualizando la posición de la pelota usando esta función cada vez, y haciendo esto de manera continuada conseguiremos ir reduciendo C hasta que (en teoría) se llegue hasta el mínimo global.



¿Y cómo podemos aplicar el gradiente descendente para aprender en una red neuronal? Pues de la misma manera que en lo explicado anteriormente al ir variando ***v*** se consigue que la bola ruede hasta el fondo del valle, en la red lo que tenemos que conseguir es que los pesos y ***bias*** funcionen de la misma manera y conseguir así el mínimo de la función de coste cuadrático de antes.

$$w_k \to w_k' = w_k - \eta \frac{\partial C}{\partial w_k}$$
$$b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l}.$$

Para que todo esto funcione correctamente tal y como se ha comentado, hay que elegir un ***learning rate*** (***η***) que sea suficientemente pequeño, ya que si no, podemos acabar con valores de ***ΔC*** que sean mayores que 0, lo cual obviamente no sería bueno.
Al mismo tiempo, tampoco queremos que ***η*** sea excesivamente pequeño, ya que entonces el gradiente descendente avanzaría muy poco a poco y el algoritmo actuaría muy lento.
Es decir, hay que conseguir que ***η*** funcione bien en la ecuación y que sea una buena aproximación, pero que tampoco haga que el algoritmo sea excesivamente lento, veremos más tarde como conseguir esto.

Esta forma de gradiente descendiente que vamos a aplicar tiene muchas ventajas como hemos podido observar a lo largo del documento, pero tiene un gran inconveniente: resulta que calcular las segundas derivadas parciales de ***C*** puede ser muy costoso.
Para ver porque es muy costoso, vamos a plantearlo; supongamos que queremos calcular todas las derivadas parciales,

$$\partial^2 C / \partial v_j \partial v_k$$

Si hay un millón de variables ***vj*** entonces necesitamos calcular alrededor de un trillón (un millón al cuadrado) de segundas derivadas parciales, lo cual es computacionalmente muy muy costoso.

Para tratar de solucionar este problema, vamos a aplicar el llamado ***gradiente descendente estocástico***, y acelerar así el aprendizaje del programa.
La idea es estimar el gradiente de ***∇C*** calculando ***∇Cx*** para una pequeña muestra de puntos elegidos aleatoriamente, y haciendo la media de esta pequeña muestra podemos conseguir una gran

estimación del verdadero gradiente ∇C, habiendo acelerado así mucho el gradiente descendiente en sí, y por tanto el aprendizaje.

El gradiente descendente estocástico funciona seleccionando un pequeño número de *m* de inputs de entrenamiento aleatoriamente seleccionados, a los cuales marcaremos como X1, X2,..., Xm y nos referiremos a ellos como ***mini-batch***, y estimaremos el coste total del gradiente calculando los gradientes de los valores seleccionados aleatoriamente de la ***mini-batch*** en cada momento.

Para conectar esto con el aprendizaje en las redes neuronales, supongamos que ***wk*** y ***bl*** definen los pesos y los ***bias*** en nuestra red neuronal, entonces el gradiente descendiente estocástico funciona seleccionando una pequeña cantidad (***mini-batch***) de inputs de entrenamiento y empezar a entrenar con ellos,

$$w_k \to w_k' = w_k - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial w_k}$$

$$b_l \to b_l' = b_l - \frac{\eta}{m} \sum_j \frac{\partial C_{X_j}}{\partial b_l},$$

Luego se coge otra ***mini-batch*** diferente y se entrena con esos valores también, y así hasta que no quede ningún valor de entrenamiento que no se haya procesado, completando así una etapa (***epoch***) del entrenamiento, empezando entonces una nueva y mejorada.

Se puede concebir el ***gradiente descendiente estocástico*** como un sondeo político, es mucho más simple probar y trabajar sobre un pequeño grupo de datos/personas (nuestras ***mini-batch***) que aplicar el gradiente descendiente a todo el grupo de datos/personas.
Por ejemplo, si tenemos un ***dataset*** de entrenamiento de *n* = 60.000 (como el del MNIST) y seleccionamos un pequeño grupo de *m* = 10, significa que habremos conseguido acelerar la estimación del gradiente en factor de 6.000, que por supuesto que no será 100% perfecto, pero no tiene porque serlo, solo nos interesa saber en que dirección decrece *C*, lo cual significa que obtener la el cálculo exacto del gradiente no nos interesa.
De hecho, en la práctica, el gradiente descendente estocástico es muy usado y una técnica muy poderosa para enseñar y aprender en una red neuronal.

Implementando la red neuronal en *Python* para clasificar dígitos

Vamos a pasar ahora a lo interesante, lo que realmente nos atañe y el objetivo de este trabajo, escribir un programa en *Python* que aprenda a reconocer dígitos escritos a mano usando *gradiente descendente estocástico* y los datos del MNIST.

En primer lugar, la pieza angular del programa es la clase *Network*, que servirá para representar la red neuronal que vamos a trabajar, aquí a continuación está el código de cómo inicializar un objeto *Network*:

```
class Network(object):

    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]
```

En este fragmento de código, *sizes* es una lista que contiene la cantidad de neuronas de cada una de las capas de la red neuronal, por ejemplo, pongamos el caso de que se quiere crear una *Network* con 2 neuronas en la primera capa, 3 neuronas en la segunda capa, y 1 neurona en la última capa, generando una red neuronal de tres capas, se inicializará de la siguiente manera: *net = Network([2, 3, 1])*

Los *biases* y los pesos (*weights*) del objeto *Network* se inicializan de manera aleatoria al ser inicializada, haciendo uso de la función *random.randn* de la librería Numpy (necesario importarla para el proyecto, además del *random*) para generar así distribuciones Gaussianas con media de 0 y desviación estándar de 1.
Esta inicialización random le proporciona al algoritmo de nuestro *gradiente descendente estocástico* un lugar donde empezar, y aunque quizá hay otras formas de inicializar estos *biases* y pesos, esto nos sirve para el problema que vamos a tratar.
Cabe también destacar que la *Network* asume que la primera capa de neuronas es la capa de entrada (*input layer*), y omite cualquier conjunto de *biases* para esas neuronas, ya que los *biases* solo se utilizan para calcular los resultados de las capas futuras.

También cabe destacar que los *biases* y pesos se almacenan en listas de matrices de Numpy, así que por ejemplo, *net.weights[1]* es una matriz Numpy que tiene almacenados los pesos conectando la segunda y tercera capa de neuronas.
Para hacerlo más sencillo (y no poner *net.weights[1]*) , vamos a denotar la matriz de pesos como *w*, una matriz en la que w_{jk} es el peso que afecta a la conexión entre la neurona *k* de la segunda capa y la neurona *j* de la tercera capa
Aquí el lector puede pensar que el indexado de letras debería ser al revés, pero usando este orden significa que el vector de activaciones de la tercera capa sería:

$$a' = \sigma(wa + b).$$

Vamos a explicar esta ecuación, *a* es el vector de activaciones de la segunda capa de neuronas, y para obtener *a´* se multiplica *a* por la matriz de pesos *w* y el vector *b* de *biases*, y entonces aplicamos la función sigmoide a todo el conjunto *wa + b*.
Y con todo esto en mente, podemos introducir ahora la función sigmoide dentro de nuestro programa de *Python* fuera de la clase *Network*:

```
def sigmoid(z):
    return 1.0/(1.0+np.exp(-z))
```

Luego de esto, toca añadir el método *feedforward* dentro de la clase *Network*, el cual, dándole un determinado input *a* de la red neuronal, calcula y devuelve su resultado, aplicando la función vista arriba de *a´*:

```
def feedforward(self, a):
    for b, w in zip(self.biases, self.weights):
        a = sigmoid(np.dot(w, a)+b)
    return a
```

Por supuesto, lo principal que queremos que haga nuestra red neuronal es aprender, y para ello le añadiremos un método *SGD* que implementará el gradiente descendente estocástico (*Stochastic Gradient Descent*):

```
def SGD(self, training_data, epochs, mini_batch_size, eta,
        test_data=None):

    training_data = list(training_data)
    n = len(training_data)

    if test_data:
        test_data = list(test_data)
        n_test = len(test_data)

    for j in range(epochs):
        random.shuffle(training_data)
        mini_batches = [
            training_data[k:k+mini_batch_size]
            for k in range(0, n, mini_batch_size)]
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)
        if test_data:
            print("Epoch {} : {} / {}".format(j,self.evaluate(test_data),n_test))
        else:
            print("Epoch {} complete".format(j))
```

Vamos a hablar primero de las variables/argumentos que se le pasan a la función, *training_data* es una lista de tuplas (x, y) que representa los datos de entrenamiento introducidos y los resultados esperados, las variables *epochs* y *mini_batch_size* son lo que parecen, la cantidad de etapas que se entrenará y el tamaño de los grupos para realizar el muestreo y *eta* es la tasa de aprendizaje (*learning rate, η*).
Si *test_data* (argumento opcional) está presente, entonces el programa evaluará la red neuronal después de cada etapa de aprendizaje, y imprimirá el progreso parcial realizado en cada una de ellas, permitiendo así el registro del progreso y del aprendizaje, aunque ralentice un poco el proceso.

El código funciona de la siguiente manera, en cada etapa (*epoch*), se empieza mezclando aleatoriamente los datos de enterenamiento, y los divide en *mini-batches* de aproximadamente el mismo tamaño, permitiendo así una manera sencilla de muestrear los datos de entrenamiento.
Entonces, para cada *mini-batch* se aplica un único paso del gradiente descendiente, mediante *self.update_mini_batch(mini_batch, eta)*, lo que actualiza los pesos y los *biases* de la red neuronal en concordancia de una única iteración del gradiente descendente, usando solo los datos de entrenamiento de esa *mini_batch* (la bola empieza a bajar por el valle de la montaña).

Este método *update_mini_batch* se programa así dentro de nuestra *Network*:

```
def update_mini_batch(self, mini_batch, eta):
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
        delta_nabla_b, delta_nabla_w = self.backprop(x, y)
        nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b)]
        nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w)]
    self.weights = [w-(eta/len(mini_batch))*nw
                    for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb
                   for b, nb in zip(self.biases, nabla_b)]
```

Dónde la mayoría del trabajo y de la computación se realiza en la línea *delta_nabla_b, delta_nabla_w = self.backprop(x, y)*, lo cual invoca algo llamado el *backpropagation algorithm* (algoritmo de la propagación hacia atrás), una manera veloz y rápida de calcular el gradiente de la función coste.
Así, *update_mini_batch* funciona simplemente calculando estos gradientes para cada uno de los ejemplos de entrenamiento dentro de la *mini_batch*, y luego actualiza los *self.weights* y los *self.biases* apropiadamente y en consonancia de lo calculado.

El código de *self.backprop* no va a ser desarrollado en este trabajo por motivos prácticos, y es que quedaría un documento demasiado extenso (más de lo que ya lo es) y los cálculos y expresiones matemáticas que se necesitan explicar son más complejos, así que dejo aquí un link a un vídeo (no hace falta ver las partes anteriores si se ha entendido este trabajo hasta aquí) que lo explica de manera muy didáctica y sencilla para que si hay alguien interesado en su funcionamiento, pueda saciar ese conocimiento.
Para el que no quiera indagar más sobre el tema, vamos a asumir que *self.backprop* se comporta como se ha dicho, y devuelve el gradiente apropiado para el coste asociado al ejemplo de entrenamiento *x*.

Vamos a echar un vistazo ahora al programa entero, añadiendo más funciones a las ya explicadas hasta aquí.
Exceptuando *self.backprop*, el programa es bastante auto explicativo, y todo el peso y cálculos importantes se realizan en *self.SGD* y *self,update_mini_batch*, que ya han sido vistos y explicados.
Además, *self.backprop* hace uso de dos funciones extra, *sigmoid_prime*, que calcula la derivada de la función σ, y *self.cost_derivative*, que devuelve el vector de derivadas parciales para un determinado vector de activaciones (*output_activations*).

Vamos ahora a probar el programa que hemos elaborado, ¿cómo de bien reconoce los dígitos escritos a mano?
Bueno, vamos a empezar por cargar los datos del MNIST, para ello, se va a hacer uso de un pequeño programa, *mnist_loader.py* (viene dado por el MNIST para poder procesar sus datos), entonces se ejecuta en el shell de *Python*:

```
>>> import mnist_loader
>>> training_data, validation_data, test_data = \
...                 mnist_loader.load_data_wrapper()
```

Tras cargar los datos del MNIST, vamos a utilizar nuestro archivo *Network.py* creando un objeto *Network* con 30 neuronas ocultas, escribiendo en el shell de *Python* lo siguiente:

```
>>> import Network
>>> net = Network.Network([784, 30, 10])
```

El 784 se refiere a cada uno de los píxeles que hay en una imagen del MNIST (como se ha explicado anteriormente en el trabajo, 28x28) y el 10 se refiere a cada una de las neuronas de salida, cada una representa un dígito (del 0 al 9).
Tras haber creado nuestro objeto *Network*, se va a aplicar el *gradiente descendiente estocástico* para aprender de los datos del MNIST en un periodo de 30 etapas, con un tamaño de *mini-batch* de 10 y un ratio de aprendizaje de $\eta = 3.0$:

```
>>> net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

Esto tardará un poco en ejecutarse (debido al número de etapas, si se quiere agilizar el trabajo, se puede disminuir la cantidad de etapas necesarias para aprender y reducir también la cantidad de neuronas ocultas).
Una vez nuestra *Network* esté entrenada, ya no le costará nada analizar y clasificar cada dígito que se le introduzca como *input*, esta tardanza ocurre solo al aprender, como es típico en cualquier algoritmo de *machine learning*.

Mientras se va ejecutando el programa durante sus diferentes etapas se puede observar que solo en la primera etapa ya tiene alrededor de 9.000 sobre 10.000 aciertos, porque así de bueno es este método, y este valor va incrementando a medida que aumentas las etapas de entrenamiento:

```
Epoch 0 : 9028 / 10000
Epoch 1 : 9143 / 10000
Epoch 2 : 9241 / 10000
Epoch 3 : 9289 / 10000
Epoch 26 : 9458 / 10000
Epoch 27 : 9469 / 10000
Epoch 28 : 9450 / 10000
Epoch 29 : 9485 / 10000
```

Como resultado, nuestra ***Network*** consigue un porcentaje de éxito de un 95% (va variando dependiendo de la ejecución debido a la inicialización de pesos y ***biases*** aleatoria), pero se trata de un resultado muy alentador para su primer intento.

Vamos a ver ahora qué sucede si variamos el número de neuronas ocultas de 30 a 100, ejecutando exactamente el mismo código pero en este caso crearemos el objeto ***Network*** así:

```
>>> net = Network.Network([784, 100, 10])
>>> net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
```

Esto elevará los resultados a alrededor del 96-97% (aunque empiece con valores inferiores al ejemplo anterior, alcanzará mayor exactitud con más neuronas ocultas.

Conseguir estas exactitudes no ha sido fruto del azar, y es que se han tenido que ir haciendo pruebas con las épocas de entrenamiento, el tamaño de las ***mini-batches*** y el ratio de aprendizaje η, los cuales son conocidos como ***hyper-parameters*** de nuestra ***Network*** (para diferenciarlos de los pesos y los ***bias***, parámetros normales que nuestra red neuronal va a aprendiendo a darles valor).
Si estos ***hyper-parameters*** se escogen de manera pobre, no es nada complicado obtener resultados malos, como ocurre por ejemplo si elegimos de ratio de aprendizaje el valor η = 0.001:

```
>>> net = Network.Network([784, 30, 10])
>>> net.SGD(training_data, 30, 10, 0.001, test_data=test_data)
```

Se puede comprobar que los datos obtenidos son mucho menos alentadores:

```
Epoch 0 : 931 / 10000        Epoch 27 : 2470 / 10000
Epoch 1 : 965 / 10000        Epoch 28 : 2520 / 10000
Epoch 2 : 982 / 10000        Epoch 29 : 2569 / 10000
```

Aún así, se percibe cierto aprendizaje durante las etapas, lo que sugiere que aumentar el ratio de aprendizaje a, por ejemplo, 0.01, obtendremos mejores resultados, y así sucesivamente hasta llegar hasta 3.0, que es el learning rate "idílico" para nuestro proyecto según las pruebas llevadas a cabo.
Esto significa que, aunque elijamos de manera pobre los ***hyper-parameters*** de nuestra ***Network***, al menos tenemos cierta información que nos ayudará a mejorarlos y llegar a los correctos o más indicados para nuestro caso.

Pero aunque pueda parecer trivial, es un reto tratar de debuggear una red neuronal, por ejemplo imaginemos que con 30 neuronas ocultas estamos tratando de seguir aumentando el ratio de aprendizaje y en este caso hemos llegado hasta $\eta = 100.0$:

```
>>> net = Network.Network([784, 30, 10])
>>> net.SGD(training_data, 30, 10, 100.0, test_data=test_data)
```

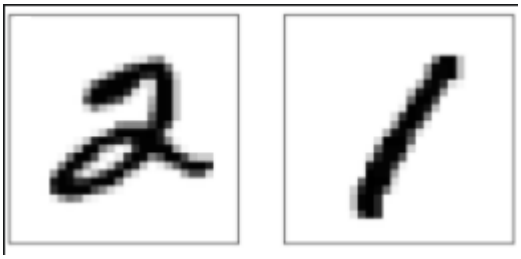
Aquí nos daremos cuenta de que hemos ido demasiado lejos, y que el ratio de aprendizaje tiene un valor demasiado elevado:

```
Epoch 0 : 892 / 10000        Epoch 27 : 892 / 10000
Epoch 1 : 892 / 10000        Epoch 28 : 892 / 10000
Epoch 2 : 892 / 10000        Epoch 29 : 892 / 10000
```

La lección a aprender aquí es que debuggear una red neuronal no es nada sencillo, y que aunque parezca programación simple, hay un cierto arte y complejidad detrás, y es que si alguien quiere obtener buenos resultados en su red neuronal o en cualquier programa de *machine learning* tiene que aprender a debuggear correctamente sus proyectos.

Así, como hemos podido comprobar, nuestro programa y nuestra red neuronal obtiene unos resultados bastante decentes pero, ¿qué significa eso? ¿Decentes comparados con qué?
Es informativo tener un programa (ninguna red neuronal) de test para comparar los resultados y entender si de verdad nuestra red neuronal está funcionando correctamente o no.
Lo más simple es tener como referencia un programa que adivina el número de manera aleatoria, aunque eso sería un ejemplo muy simple, es mejor que nada, pero vamos a ir más allá.

Vamos a probar a ver la cantidad de "oscura" que es una imagen, ya que por ejemplo, un 2 siempre tendrá un poco más de "oscuro" que un 1, ya que habrá más píxeles marcados en negro, lo cual se puede comprobar en estas imágenes:



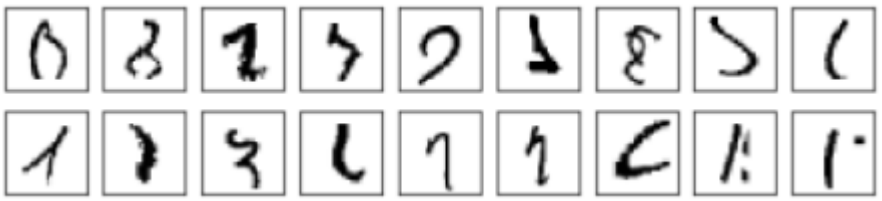
Al recibir cada nueva imagen, el programa calcula cómo de "oscura" es, y entonces tratará de adivinar y deducir qué dígito es por el promedio de píxeles oscuros que hay en la imagen.

Esto es un procedimiento muy simple, y resulta sencillo de programar, así que no se va a explicar todo el procedimiento, simplemente se va a añadir el programa como *mnist_average_darknes.py*, y se obtendrá un resultado de 2.225 sobre 10.000, es decir, una exactitud de un 22.25%.

Tampoco es muy difícil conseguir exactitudes de entre 20 y 50 por ciento, o incluso superar este 50 por ciento.
Vamos a probar para ello, usar uno de los algoritmos más conocidos, el algoritmo *SVM* o el *support vector machine* (si no se conoce este algoritmo no pasa nada, tampoco hará falta saber los detalles de dicho algoritmo para entender esta aplicación), pero en vez de aplicar este algoritmo usaremos *scikit-learn*, una librería de *Python* que proporciona una librería para *SVM* conocida como *LIBSVM*.

Si ejecutamos el clasificador de *scikit-learn* haciendo uso de los ajustes por defecto, obtenemos una exactitud de 93-94 por ciento (el código se encuentra en *mnist_svm.py*, lo cual ya es una gran mejora respecto a nuestro programa tan simple de antes.
De hecho, esto significa que el algoritmo *SVM* está funcionando casi tan bien como las redes neuronales, y cambiando sus parámetros es posible hacer que su rendimiento llegue al 98.5% de exactitud, lo que significa que este algoritmo con los parámetros bien elegidos y adaptados, solo falla un dígito cada 70, lo cual está muy muy bien, pero ¿pueden las redes neuronales hacerlo mejor?

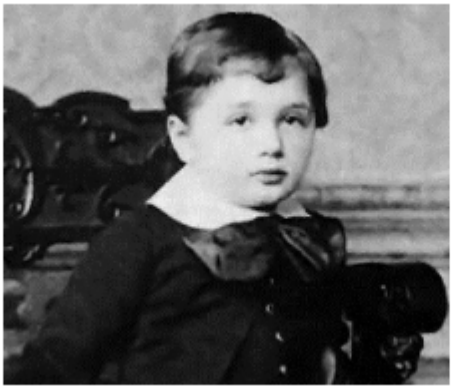
La respuesta es que sí, las redes neuronales actualmente desbancan cualquier otra forma de clasificar los datos del MNIST, incluyendo el algoritmo *SVM*, y el récord actual se encuentra en la correcta clasificación de 9.979 dígitos de 10.000, lo cual alcanza un nivel muy equivalente al de los seres humanos, incluso puede llegar a ser discutiblemente mejor, ya que algunas imágenes del MNIST pueden resultar complicadas de identificar incluso para los humanos, como por ejemplo:



Y para obtener esta exactitud sólo se han realizado algunas mejoras en el programa presentado en este trabajo como ejercicio, lo que demuestra un simple algoritmo de aprendizaje y buenos datos de entrenamiento superan muchas veces a un algoritmo muy sofisticado.

EPÍLOGO: HACIA EL "DEEP LEARNING"

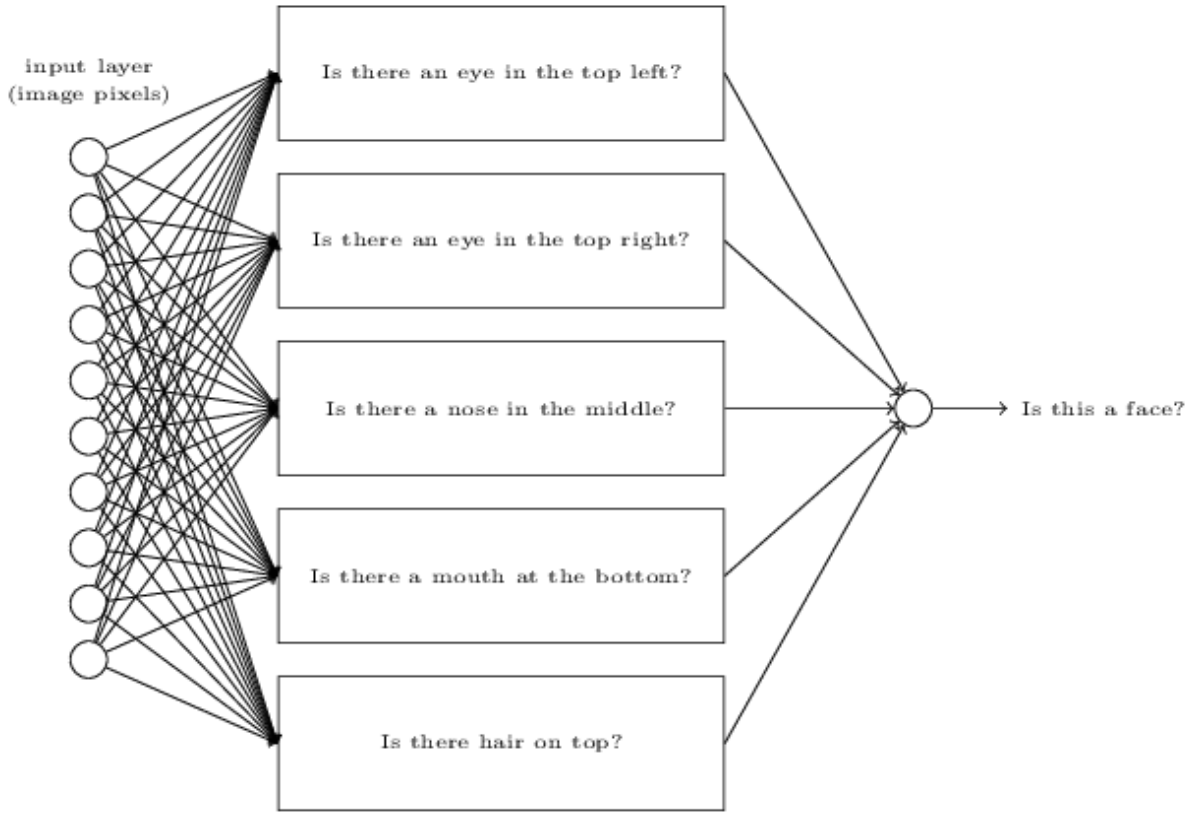
Para finalizar este trabajo, vamos a volver hacia atrás en la interpretación que se le dió a las neuronas artificiales al inicio de todo, como herramientas para sopesar evidencias y datos, imaginemos que queremos determinar si una imagen muestra un rostro humano o no:


Podemos abordar este problema de la misma manera que hemos abordado el problema de reconocer dígitos escritos a mano, usando los píxeles de la imagen como datos de entrada (***input***) y que la red calcule y obtenga como resultado (***output***) si efectivamente "Es una cara" o si "No, no es una cara".

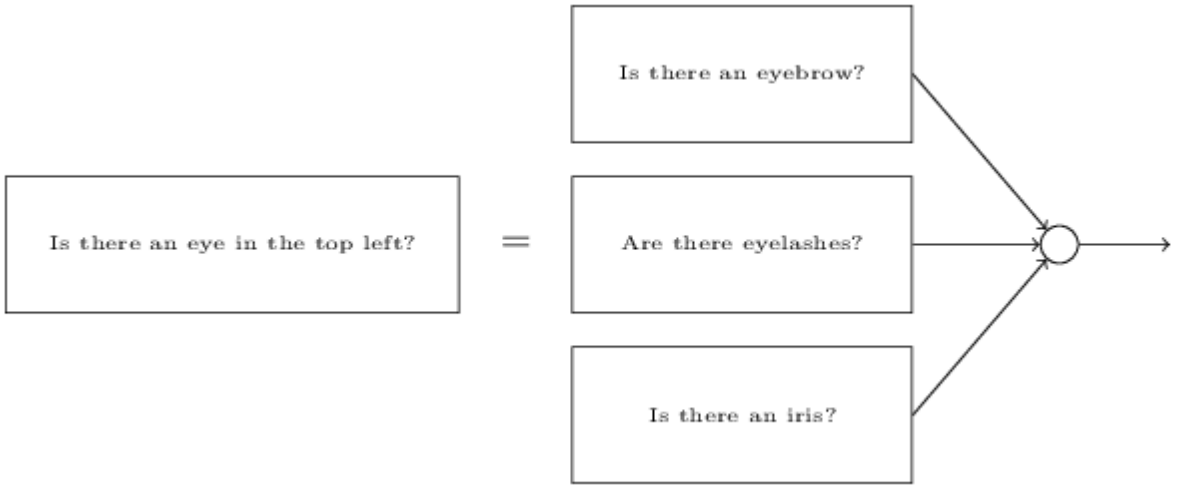
Supongamos que hacemos esto, pero sin utilizar un algoritmo de aprendizaje, y usamos la siguiente heurística: "¿Tiene la imagen un ojo en la parte superior izquierda?", "¿Y en la parte superior derecha?", "¿Hay una nariz enmedio?", "¿Tiene una boca en la parte central inferior de la imagen y pelo en la parte central superior?"...
Si la respuesta a muchas de estas preguntas es "sí", o "probablemente o parcialmente si", entonces la red concluirá que se trata de una cara, y viceversa, si muchas respuestas son "no", entonces determinará que no se trata de un rostro humano lo que muestra la imagen.

Por supuesto, esto es una heurística un poco "naive" y sufre de muchas deficiencias (la persona puede ser calva, y por tanto no tiene pelo, puede que se esté mostrando solo parte de la cara, o que dicha cara esté en un determinado ángulo...) pero esta heurística sugiere que si podemos construir redes neuronales para responder a cada una de las preguntas anteriores, entonces también podemos combinar esas redes neuronales para tratar el problema de la detección facial.
A continuación añado una posible arquitectura, con rectángulos para cada una de las sub redes neuronales (no se trata de un acercamiento realista del problema, solo nos va a ayudar a intuir mejor cómo funcionan las redes neuronales).



Entonces vamos a imaginar que entramos dentro de la primera subred neuronal, "¿Hay un ojo en la parte superior izquierda?", lo cual se puede descomponer en otras preguntas como, por ejemplo, "¿Hay una ceja?", "¿Hay pestañas?", "¿Tiene un iris en el centro?"..., y aunque estas preguntas también dependen de diferentes situaciones y posiciones, vamos a mantener así las sentencias simples en vez de "¿Hay una ceja en la parte superior izquierda y además está encima de las pestañas y del iris?", para mostrar el problema de manera más sencilla.

Ahora sabemos pues que podemos dividir esta subred neuronal en lo siguiente:



Estas preguntas también se pueden subdividir muchos más veces en múltiples capas, y el resultado final serán subredes cuya pregunta se puede responder con simples píxeles individuales.
Estas preguntas comprobarán, por ejemplo, la presencia o ausencia de formas simples en determinados puntos de la imagen, preguntas que pueden responderse con neuronas simples conectadas a los mismos píxeles de la imagen.

El resultado final son redes que dividen problemas muy complejos (como el de si aparece una cara en la imagen o no) en problemas muy simples que pueden ser respondidas a nivel de píxeles individuales, y esta simplificación ocurre a través de diferentes capas, con las primeras respondiendo preguntas muy sencillas y específicas sobre la imagen y las últimas y más avanzadas que construyen una jerarquía de conceptos más complejos y abstractos.
Estas redes neuronales con este tipo de estructura con múltiples capas (dos o más capas de neuronas ocultas) se llaman ***deep neural networks***.

Y desde 2006, se ha ido desarrollando un conjunto de técnicas que permiten el aprendizaje en las ***deep neural networks***, y este aprendizaje se basa en el gradiente descendiente estocástico y en la ***backpropagation***, aunque también incluyen otras ideas.
Estas técnicas juntas han permitido entrar redes mucho más grandes y con mayor profundidad de capas a poder ser entrenadas (ahora de normal se entrenan redes neuronales de 5 a 10 capas ocultas), y resulta que estas trabajan y aprenden diferentes problemas que redes neuronales simples que tienen, por ejemplo, una o dos capas ocultas.
La razón de esto es, por supuesto, la habilidad de las ***deep neural networks*** para construir una jerarquía de conceptos, y comparando estas redes con redes más simples es como comparar un lenguaje de programación con la habilidad de poder llamar a funciones con un lenguaje de programación "desnudo" sin la habilidad de hacer esas llamadas, y es que la abstracción en las redes neuronales funciona de manera diferente a como lo hace en la programación convencional, pero es igual de importante en ambos casos.