Universidad Nacional del Centro de la Provincia de Buenos Aires
Facultad de Ciencias Exactas

Redes Neuronales Artificiales

Ariel E. Repetur

Trabajo final presentado para obtener el título de
Licenciado en Ciencias Matemáticas.

Director: Pablo A. Lotito
Codirector: Guido R. Báez

Tandil, Abril de 2019

Índice

Prefacio

Es sabido que existen ciertos problemas que resultan difíciles de resolver para los humanos pero relativamente accesibles para las computadoras. Esta clase de problemas usualmente se pueden describir por una lista de reglas formales, como ocurre por ejemplo en el ajedrez. De hecho, en los primeros días de la inteligencia artificial se atacaron y resolvieron con éxito varios de estos problemas, logrando con frecuencia un desempeño muy superior al que una persona experta en la tarea podría alcanzar. No obstante, el desafío principal para la inteligencia artificial resultó ser la resolución de tareas que nos resultan fáciles de realizar pero difíciles de especificar formalmente, como puede ser el reconocimiento del habla o del contenido en imágenes.

Las limitaciones de los sistemas formales han hecho que se busquen otras alternativas, y una solución a este problema ha sido el desarrollo de métodos que les permitan a las computadoras adquirir su propio conocimiento a partir de los datos en bruto. Estas técnicas constituyen el dominio de lo que se conoce como *machine learning*, y su introducción ha permitido que las computadoras sean capaces de atacar problemas que involucran conocimiento del mundo real y tomar decisiones que parecen subjetivas, como puede ser decidir si un correo electrónico es spam o no.

El rendimiento de muchos de estos algoritmos depende de la representación de los datos que se les brinda. Uno puede tener un sistema de inteligencia artificial que realiza diagnósticos sobre pacientes pero que no los examina de manera directa, sino que decide en base a ciertas piezas de información que sirven para representar el estado del paciente. Estos datos se conocen como *características* y lo que hace el algoritmo es aprender como se relacionan estas características con varios resultados.

Sin embargo, para muchas tareas resulta difícil decidir qué características deberían ser extraídas. Por ejemplo, supongamos que uno quisiera detectar un avión en una fotografía. Sabemos que los aviones tienen alas, por lo que podríamos considerar la presencia de alas como una característica. El problema es que es difícil describir como luce el ala de un avión en función de los valores de los píxeles. La forma geométrica es simple, pero la tarea se dificulta por la presencia de sombras o de objetos que obstruyen parte de la imagen, las variaciones en el ángulo de visión, el brillo causado por el sol, etc.

Dado que puede ser muy difícil extraer características de alto nivel a partir de los datos en bruto, hay enfoques que consisten en permitir que la computadora pueda construir representaciones más complejas a partir de otras más simples. Por ejemplo, uno puede entrenar un modelo de machine learning para realizar una de las operaciones más básicas en el análisis de imágenes, que es la detección de bordes. A partir de esta representación de la imagen en términos de sus bordes, uno puede entrenar otro modelo que utilice esa información para identificar esquinas y contornos. Luego, otro modelo usa las características anteriores para identificar la presencia de determinados objetos, y así sucesivamente. Este es el enfoque utilizado en *deep learning* y el ejemplo por excelencia de este tipo de modelo es la *red neuronal feedforward*, también conocida como *perceptrón multicapa*.

En este trabajo comenzaremos por definir un tipo de neurona artificial, el perceptrón, y luego veremos como se puede adaptar para construir el perceptrón multicapa. Como paso siguiente se planteará el entrenamiento como un problema de optimización e introduciremos dos algoritmos esenciales: uno es el descenso por gradiente estocástico, una adaptación del descenso por gradiente clásico para el contexto de redes neuronales; el

otro es el algoritmo de backpropagation, muy importante en deep learning, necesario para calcular de manera eficiente los gradientes. En el proceso veremos un problema clásico de redes neuronales, que es la clasificación de dígitos escritos a mano, para lo cual introduciremos un conjunto de datos denominado MNIST. Luego implementaremos una red neuronal básica, entrenada sobre este conjunto, que resuelve el problema de clasificación antes mencionado.

En secciones posteriores se examinarán maneras de mejorar el aprendizaje, para lo cual será necesario explorar cuestiones relacionadas a la arquitectura de la red y al planteo del problema de optimización. Se discutirán la elección de la función costo y distintos métodos de inicialización de los parámetros de la red, como también la manera en que distintos tipos de unidades (neuronas) influyen en el entrenamiento. Algunas de estas mejoras serán implementadas y aplicadas a nuestra red básica, comparando los resultados con los obtenidos previamente.

Al entrenar redes neuronales no interesa tanto el rendimiento sobre el conjunto de entrenamiento, sino que lo que importa es que la red sea capaz de aplicar el conocimiento adquirido durante el entrenamiento a nuevos datos. A esta habilidad se la conoce como generalización y existen técnicas destinadas a mejorar esta capacidad. En su conjunto estas técnicas reciben el nombre de regularización. Algunos de estos métodos serán explorados en la sección 3.

En la sección 4 se hará un repaso por algunos de los métodos de optimización que se utilizan habitualmente en redes neuronales. Estos se basan en el algoritmo de descenso por gradiente estocástico, sobre el cual se aplican distintas modificaciones a fin de mejorar su rendimiento.

Por último veremos una de las arquitecturas de red neuronal que más éxito ha tenido en los últimos años: la red convolucional. Se discutirá la motivación y las técnicas sobre las que se fundamenta. Luego se verá como realizar una implementación basada en los frameworks utilizados actualmente en deep learning y se evaluarán los resultados.

1. Introducción a las Redes Neuronales

1.1. Perceptrón

Muchos problemas consisten en la toma de decisiones. Tomar decisiones es inevitable y sin embargo no es algo necesariamente fácil de realizar, particularmente cuando uno espera tomar una decisión informada y precisa, o en otras palabras, una decisión basada en la información y criterios de los que uno dispone, capaz de responder adecuadamente al problema planteado.

En el proceso de decisión es probable que uno considere ciertas piezas de información como más importantes que otras, más decisivas, o en definitiva, que tienen mayor *peso*. También es necesario estar lo suficientemente seguro como para tomar la decisión; el grado de convencimiento necesario para tomar una decisión u otra puede variar de acuerdo al problema o la persona, pero en cualquier caso se requiere superar cierto *umbral* de certidumbre para decidir por A en vez de B.

Como ejemplo, podemos pensar en la siguiente situación: una persona, a la que llamaremos D–, tiene que decidir si mantiene o renuncia a su trabajo. En esa decisión hay varios factores que sopesar. Entre estos factores podríamos considerar el grado de satisfacción con su empleo, el salario, la estabilidad laboral, etc. Denotaremos con x_1, x_2, x_3 a los factores antes mencionados y vamos a suponer que podemos cuantificarlos con números enteros, donde un valor más alto representa una mejor valoración. Supongamos que D– tiene un empleo que aborrece, por el cual recibe un salario aceptable aunque no muy alto, pero que por otra parte le brinda una gran estabilidad laboral. Podríamos representar la situación asignando $x_1 = -1$, $x_2 = 1$, $x_3 = 2$. Por otra parte, para D– la estabilidad laboral pesa mucho en esta decisión, más que su disconformidad con el empleo, mientras que el salario influye pero no demasiado. Esto se puede representar con pesos $w_1 = 2$, $w_2 = 1$ y $w_3 = 3$. Ahora vamos a suponer que el umbral de conformidad que tiene que superar para mantener su trabajo es $T = 2$. Lo que haremos es considerar una suma pesada de los distintos factores, y si esta supera el umbral, entonces diremos que D– tiene suficientes motivos para mantener su trabajo, mientras que en caso contrario debe renunciar.

$$\sum_{i=1}^{3} w_i x_i = 2 \cdot (-1) + 1 \cdot 1 + 3 \cdot 2 = 5 > 2. \tag{1.1}$$

Y resulta que D– debe mantener su trabajo.

Por otra parte, podría ser que D– tuviera una valoración distinta de la situación. Podría ser que esté buscando un empleo que realmente sienta que se justifique. Para reflejar eso podríamos asignar $T = 10$. En ese caso vemos que de acuerdo al criterio anterior, la decisión cambia, y D– debería renunciar. Por el contrario, si asignamos $T = -10$ estaríamos modelando una situación en la que todos los factores tienen que ser muy negativos como para justificar la renuncia.

Otra forma en la que la valoración de la situación cambia es si los pesos son distintos. Si para D– el estar realizando un trabajo con el que se encuentre a gusto pesa más que todo lo demás, podríamos reflejarlo asignando $w_1 = 10$. En ese caso,

$$\sum_{i=1}^{3} w_i x_i = 10 \cdot (-1) + 1 \cdot 1 + 3 \cdot 2 = -3 < 2. \tag{1.2}$$

En este contexto se puede ver que basta con algunos disgustos para motivar la renuncia.

En términos matemáticos, lo que acabamos de hacer es construir una función que acepta varias entradas $x_1, \ldots, x_n$, las cuales cuantifican las distintas características que uno modela en el problema, para luego devolver una salida binaria que representa la decisión, de acuerdo a la función:

$$f(x_1, \ldots, x_n) = \begin{cases} 0 & \text{si } \sum_{k=1}^{n} w_i x_i \leq T, \\ 1 & \text{si } \sum_{k=1}^{n} w_i x_i > T; \end{cases} \tag{1.3}$$

donde w_i es el *peso* que cuantifica cuan relevante es la característica x_i y T es el *valor de umbral*. Si a las características y a los pesos los representamos por medio de vectores x y w, podemos reescribir lo anterior como

$$f(x) = \begin{cases} 0 & \text{si } w \cdot x \leq T, \\ 1 & \text{si } w \cdot x > T. \end{cases} \tag{1.4}$$

Ahora podemos hacer $b = -T$, con lo cual llegamos a una expresión lineal estándar:

$$f(x) = \begin{cases} 0 & \text{si } w \cdot x + b \leq 0, \\ 1 & \text{si } w \cdot x + b > 0. \end{cases} \tag{1.5}$$

Finalmente, usando la función de Heaviside, podemos expresar lo anterior como

$$f(x) = \mathrm{H}(w \cdot x + b). \tag{1.6}$$

A esto último se lo conoce como perceptrón, y es un ejemplo de *neurona artificial*.

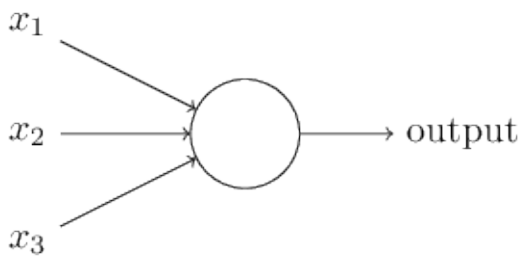


Figura 1: Perceptrón.

Hasta ahora lo que tenemos es un solo perceptrón, que toma múltiples entradas y devuelve una salida binaria, pero podemos combinar múltiples perceptrones, cada uno tomando distintas decisiones basadas en la entrada, para así obtener múltiples salidas. Si llamamos a cada perceptrón $f_1, \ldots, f_n$, lo que tenemos entonces es

$$\begin{aligned} f(x) &= (f_1(x), \ldots, f_n(x))^T \\ &= (\mathrm{H}(w_1 \cdot x + b_1), \ \ldots \ , \mathrm{H}(w_n \cdot x + b_n))^T \\ &= \mathrm{H}(w_1 \cdot x + b_1, \ \ldots \ , w_n \cdot x + b_n)^T \\ &= \mathrm{H}(Wx + b). \end{aligned} \tag{1.7}$$

donde $W = (w_1, \ldots, w_n)^T$ es la matriz que tiene a los vectores de pesos w_i como filas y $b = (b_1, \ldots, b_n)^T$.

Podemos pensar el conjunto de estos perceptrones como una *capa* que recibe un vector de entrada y devuelve un vector de salida. Un vistazo a la situación nos permite ver que ese vector de salida puede ser usado como entrada para otra capa de perceptrones, que decida en base al procesamiento realizado por la capa anterior. Si x es la entrada, f_1 es la primer capa y f_2 la segunda capa, lo que tenemos no es más que la composición de funciones $f(x) = f_2(f_1(x))$, en lo que se conoce como una red neuronal, por la manera en que se interconectan los perceptrones. La combinación de estas distintas capas nos permite realizar procesamiento a distintos niveles, para así obtener patrones de decisión más complejos. Obviamente no hay, en teoría, nada que nos impida utilizar más capas, y el número de capas que utilicemos determinará la *profundidad* de la red. Esto nos lleva cerca del concepto del perceptrón multicapa, aunque antes hay algunos inconvenientes que resolver.

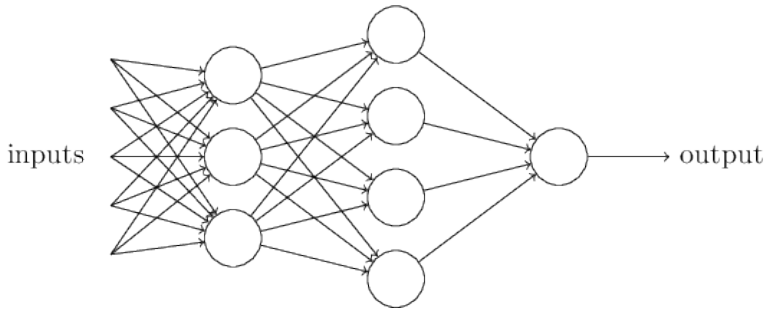


Figura 2: Red totalmente conexa con varias capas.

En la discusión del ejemplo anterior nosotros manipulamos los pesos manualmente para que el modelo pueda reflejar adecuadamente la situación. Sin embargo, es claro que para modelos complejos eso se puede volver extremadamente complicado. Para ver esto, pensemos en otro tipo de problema de decisión. Supongamos que tenemos una carta, escrita a mano. Nosotros somos capaces de recorrer la carta, aislar los trazos e ir decidiendo a qué carácter corresponde cada trazo. Dependiendo de la caligrafía del emisor esa tarea puede ser más o menos difícil. Ahora, ¿qué ocurre si queremos que una computadora realice la misma tarea? En este ejemplo tenemos al menos dos problemas distintos: uno es la segmentación, que consiste en separar las imágenes en varias partes (que en este caso pueden ser letras o palabras), mientras que el otro es la clasificación, que consiste en procesar cada una de las partes para luego asignarles un rótulo o categoría. Para simplificar el ejemplo nos vamos a limitar al problema de clasificación, ya que este será el que nos va a interesar más adelante. Para atacarlo podemos usar lo que hemos desarrollado hasta ahora.

Lo que queremos es construir una red f que decida a qué letra del alfabeto corresponde una imagen x. Para esto podemos comenzar con una red de una sola capa con 27 perceptrones, $f = (f_1, \ldots, f_{27})^T$, donde cada uno de los perceptrones f_k decide si la entrada corresponde o no a una letra individual del alfabeto. La salida deberá ser un vector con un uno en la posición de la letra que es asignada a la imagen y cero en las demás posiciones. En otras palabras, si el clasificador decide clasificar la imagen como una 'b'

entonces el vector de salida tendrá un uno en la segunda posición y ceros en las demás[1]. La entrada consistirá en una imagen digitalizada de un carácter de la carta (estamos asumiendo que la segmentación ya fue realizada). Esa imagen se representa por medio de una matriz o tensor cuyos valores corresponden a las intensidades de cada píxel. Con una matriz de $m \times n$ podemos representar una imagen en blanco y negro, mientras que con un tensor de $3 \times m \times n$ podemos representar una imagen a color con 3 canales, uno por cada color. En lo que sigue vamos a suponer que la imagen se encuentra en blanco y negro, por simplicidad. Otra representación se obtiene si concatenamos las filas o las columnas de la matriz anterior, lo que nos permite pensar la imagen como un vector. Esta última será la que usaremos a continuación.

Para que el clasificador funcione correctamente tenemos que asignar valores a la matriz de pesos W y al vector b de tal manera que podamos ponderar adecuadamente la información que tenemos, que en este caso son las intensidades de los píxeles. El problema que surge es bajo qué criterio deberían asignarse esos pesos. Supongamos que la imagen del carácter es de 20×20, con lo cual la entrada es un vector de 400×1. En ese caso W será una matriz de 27×400, mientras que b será de 27×1, por lo que tendremos que ajustar 10827 parámetros para que el modelo funcione. Se hace evidente que realizar esto manualmente y por simple inspección no parece muy razonable. Lo que necesitamos es una manera de que nuestro modelo asigne de manera automática esos valores. Dado que el funcionamiento correcto del clasificador depende de la correcta asignación de los pesos, podemos decir que lo que estamos buscando no es más que una manera de que nuestro clasificador *aprenda* a realizar correctamente su tarea. La pregunta entonces es como hacemos para *entrenarlo*.

1.2. Entrenamiento: una primera aproximación.

Para poder realizar el entrenamiento de la red, una de las primeras cosas que necesitamos es una forma de evaluar su rendimiento. Es evidente que el rendimiento en este caso está basado en la frecuencia con la que la red clasifica correctamente los caracteres escritos a mano. Si uno prueba con cien imágenes de caracteres y ve que solo clasifica correctamente diez de las cien, eso nos haría pensar que no está funcionando correctamente; por otra parte, si acertara en las cien, eso nos daría más confianza respecto al rendimiento. Implícito en lo anterior se encuentra el hecho de que necesitamos un *conjunto de entrenamiento*, compuesto de imágenes previamente rotuladas que esperamos que la red aprenda a clasificar correctamente, y con el cual vamos a evaluar su rendimiento durante el entrenamiento. También se desprende que la función que podríamos usar para cuantificar el rendimiento es el número de aciertos sobre el total de elementos en el conjunto de entrenamiento, a la que llamaremos *exactitud*. El objetivo entonces sería ajustar los parámetros w y b a fin de maximizar la exactitud. El problema con este enfoque es que la función exactitud no es suave. Pequeños cambios en los parámetros w y b en la mayoría de los casos no van a ocasionar ningún cambio en la exactitud, con lo cual resulta difícil descubrir qué ajustes realizar para mejorar el rendimiento. Se hace necesario buscar una

[1] Al menos esa sería la situación ideal, porque de momento no hay nada que evite que un perceptrón decida que la imagen corresponde a la letra 'l' y otro decida que corresponde a la letra 'i', con lo cual tendríamos varios unos en el vector. Vamos a ignorar esto por el momento, ya que más adelante vamos a usar otro enfoque que no posee este inconveniente.

alternativa.

Otra forma de cuantificar el rendimiento sería medir de alguna manera la discrepancia entre la salida de la red $f(x)$ y el valor que realmente le corresponde a x. A esta cantidad la llamamos *pérdida* o *costo* y nos dice cuan lejos estamos de clasificar correctamente la entrada. Si llamamos C_x al costo en que incurre la red f al clasificar x, y θ a los parámetros, el costo promedio es

$$C(\theta) = \frac{1}{n} \sum_x C_x(\theta), \tag{1.8}$$

y lo que nos va interesar entonces será buscar los parámetros θ que minimizan dicho costo.

Una posibilidad para cuantificar el costo en x sería

$$C_x(\theta) = \|f(x,\theta) - y(x)\|_1 , \tag{1.9}$$

donde denotamos con $y(x)$ al rotulo que le corresponde a x. Otra posibilidad, tal vez más habitual, es usar

$$C_x(\theta) = \|f(x,\theta) - y(x)\|_2 . \tag{1.10}$$

La expresión anterior se suele reemplazar por otra similar,

$$C_x(\theta) = \frac{1}{2} \|f(x,\theta) - y(x)\|_2^2 , \tag{1.11}$$

que es más conveniente al momento de diferenciarla. Si ahora calculamos el valor medio obtenemos

$$C(\theta) = \frac{1}{2n} \sum_x \|f(x,\theta) - y(x)\|_2^2 . \tag{1.12}$$

La formula (1.12) se conoce como costo cuadrático, y es diferenciable respecto a θ si f lo es, lo cual nos solucionaría el problema, si no fuera que de lo trabajado hasta ahora se puede ver que f no es diferenciable.

Recordemos que el perceptrón está definido como

$$f_i(x) = \mathrm{H}(w_i \cdot x + b_i), \tag{1.13}$$

por lo que nuestra capa de perceptrones tiene la forma

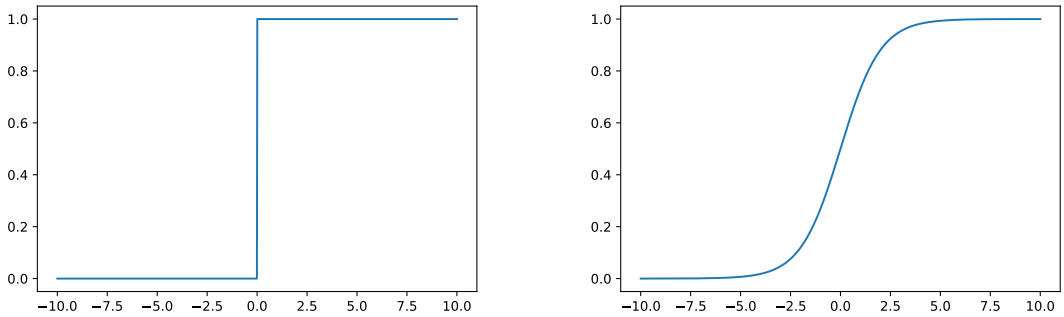
$$f(x) = \mathrm{H}(Wx + b). \tag{1.14}$$

La función de Heaviside es discontinua en cero y eso implica que pequeños cambios en los parámetros w y b pueden causar cambios bruscos en las decisiones que toman los perceptrones; una leve modificación puede cambiar completamente la clasificación. Eso hace que el proceso de aprendizaje sea difícil de controlar, al igual que nos ocurre si la función que cuantifica la pérdida no es suave. Otro problema con la función de Heaviside es que nos limita a un esquema muy rígido: los perceptrones solo pueden decidir de manera binaria y no admiten la posibilidad de trabajar con distintos grados de certeza en la clasificación. Por estos motivos se hace conveniente pensar en una alternativa antes de seguir adelante.

1.3. Neuronas Sigmoides

La función de Heaviside puede ser pensada en este contexto como una *función de activación*. El perceptrón pondera los datos por medio de $z(x) = w \cdot x + b$ y se activa o no de acuerdo a $\mathrm{H}(z)$. Esa función de activación puede ser reemplazada por otras con mejores propiedades, y en nuestro caso, preferiríamos tener una función diferenciable. Lo que podemos hacer es tomar una aproximación suave de la función de Heaviside. A continuación vamos a definir la sigmoide, o curva logística:

$$\sigma(x) = \frac{\exp(x)}{1 + \exp(x)} = \frac{1}{1 + \exp(-x)}. \tag{1.15}$$



(a) Heaviside (b) Sigmoide

Figura 3: Activaciones

Si ahora usamos esta función de activación para definir

$$f(x) = \sigma(w \cdot x + b), \tag{1.16}$$

tenemos que f es algo semejante a un perceptrón suavizado, que no se activa de manera binaria, sino que presenta un continuo de posibles grados de activación.

Por ejemplo, dado un perceptrón capaz de decidir si su entrada corresponde o no a la letra 'a', la versión suavizada nos dirá cuan cerca está la imagen de corresponder a esa letra. De hecho, podemos pensar a $f(x)$ como la probabilidad de que la imagen en cuestión sea la letra 'a'. Este cambio en la función de activación nos permite obtener cambios graduales en la salida a partir de ajustes graduales en los parámetros w y b, haciendo posible el entrenamiento de la red. También nos permite obtener más información a partir de la salida que la que una respuesta binaria puede brindarnos. A este perceptrón suavizado se lo denomina neurona sigmoide, o unidad sigmoide (los términos neurona y unidad se usan de manera intercambiable).

1.4. Perceptrón Multicapa

Como vimos al principio, es posible combinar varios perceptrones en una capa de procesamiento, con lo cual obtenemos una función $f : \mathbb{R}^m \to \mathbb{R}^n$, donde m es el tamaño

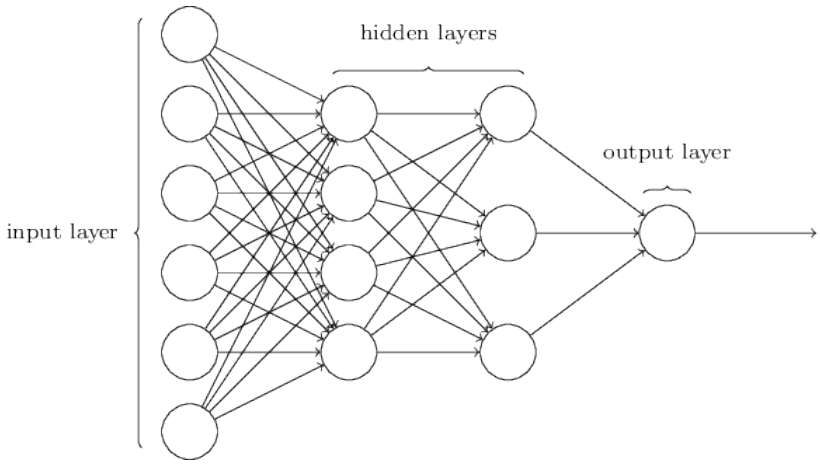


Figura 4: Perceptrón multicapa con dos capas ocultas.

de la entrada y n es el número de perceptrones, que a su vez determinan el tamaño de la salida. Dada una entrada x, la función f procesa dicha entrada mediante

$$f(x) = \phi(Wx + b), \tag{1.17}$$

donde ϕ es la función de activación que elegimos usar para esa capa, W es la matriz de pesos, y b es el vector que contiene los sesgos. También vimos que podemos componer varias de estas capas, con lo cual obtenemos una *red*, donde los perceptrones de una capa reciben como entrada a los de la capa anterior y alimentan a los de la capa que sigue. Este tipo de redes se denominan *feedforward*, porque el flujo de información va de una capa a la que sigue, sin bucles de retroalimentación. Originalmente usamos la función de Heaviside como activación, dando lugar a los perceptrones propiamente dichos. Sin embargo, hemos visto que es más conveniente usar unidades sigmoides, en vez de perceptrones, cuya activación es $\sigma(x) = \frac{1}{1+\exp(-x)}$. A las redes compuestas por varias capas de unidades sigmoides se las denomina perceptrón multicapa (esto pese al hecho de que no usan perceptrones realmente), o MLP por sus siglas en inglés, y el número de capas determina la *profundidad* de la red. También es posible, y hasta deseable, usar otras activaciones, lo cual se verá más adelante.

Dada una red f y una entrada x, a la salida $f(x)$ se la denomina, naturalmente, *capa de salida*, mientras que a la entrada x se la suele denominar *capa de entrada*, aunque no sea realmente una capa de la red. Cuando tenemos una red con varias capas (la de entrada no cuenta), todas las capas que no son de salida se denominan *capas ocultas*. Una explicación para el nombre es que la salida de las capas ocultas es usada como entrada para la siguiente capa, por lo que no se "ven" los valores que entrega; solo la capa de salida es visible (y en todo caso la de entrada también). Otra explicación es que los datos de entrenamiento nos dicen como debe comportarse la capa de salida, pero no nos dicen nada de como deben comportarse las capas intermedias; el algoritmo de entrenamiento es el responsable de decidir como utilizar esas capas para producir la salida deseada. Dado que desde ese punto de vista las capas intermedias no están directamente bajo nuestro control, tiene sentido llamarlas capas ocultas.

También vimos que la red f depende de parámetros w y b, que en su conjunto vamos a denotar con θ, y estará formada por composición sucesiva de varias capas f^i, donde el

superíndice indica el número de capa. Cada capa cuenta con sus propios parámetros, y procesa la información que recibe de acuerdo a la función

$$f^i(a^{i-1}, \theta^i) = f^i(a^{i-1}, w^i, b^i) = \sigma(W^i a^{i-1} + b^i), \tag{1.18}$$

donde a^{i-1} es la salida, o activación, de la capa anterior, W^i es la matriz de pesos de la capa i y b^i es el vector de sesgos.

El objetivo de una red feedforward es aproximar una función g. En el caso del problema de clasificación estamos asumiendo que existe una relación entre los datos que queremos clasificar y las categorías de las que disponemos, de tal manera que a cada entrada x le corresponda una categoría y. Esa relación la podemos representar por medio de una función g, tal que $g(x) = y$. Siguiendo el ejemplo que hemos trabajado hasta ahora, x puede ser un carácter escrito a mano, mientras que y es el nombre del carácter. Lo que queremos lograr con la red es poder clasificar adecuadamente esos caracteres, que no es más que pedir que la red se comporte como g. Obviamente a g no la conocemos completamente, porque sino ya tendríamos resuelto el problema de clasificación, sin mencionar que la cantidad de datos es potencialmente infinita y no podríamos clasificarlos a todos. Sin embargo, podemos considerar una muestra de información previamente rotulada, que es a lo que denominamos conjunto de entrenamiento, y entrenar la red para que aprenda a clasificar correctamente ese conjunto, con la esperanza de que eso nos sirva para generalizar a otros datos. Esto se puede ver como un problema de interpolación. Lo que estamos haciendo es aproximar g por medio de f, a partir del conocimiento de una cantidad limitada de puntos de g. Luego esperamos que f se comporte de manera aceptable para el resto de puntos. También se puede ver como un problema de extrapolación, ya que podría ocurrir que tuviéramos una observación que no fue contemplada por el conjunto de entrenamiento. No obstante, lo que esperamos es que la información colectada a partir de los datos que poseemos sea suficiente para extraer información de este nuevo caso.

En la siguiente sección vamos a retomar con el entrenamiento de la red.

1.5. Entrenamiento

Recapitulando, habíamos mencionado antes que el entrenamiento de la red involucra un problema de optimización. Específicamente, lo que queremos es minimizar la función costo (también llamada función pérdida), la cual penaliza de alguna manera los errores que comete la red al clasificar. La función costo es la esperanza sobre los costos individuales en los que incurre la red al clasificar cada elemento del conjunto de entrenamiento. Si llamamos $C_x(\theta)$ al costo para cada elemento x del conjunto, θ a los parámetros de la red, y n es el número de elementos en el conjunto de entrenamiento, entonces nuestro problema se puede expresar como

$$\arg\min_{\theta} C(\theta) = \arg\min_{\theta} \frac{1}{n} \sum_{x} C_x(\theta). \tag{1.19}$$

En otras palabras, lo que queremos es ajustar los parámetros de la red a fin de minimizar los errores de clasificación.

También habíamos visto que una posibilidad para la función costo es utilizar lo que se denomina costo cuadrático, con lo cual el problema anterior nos queda en la forma

$$\arg\min_{\theta} L(\theta) = \arg\min_{\theta} \frac{1}{n} \sum_{x} \|f(x,\theta) - y(x)\|_2^2 . \tag{1.20}$$

Más adelante usaremos otras funciones, pero como ejemplo concreto podemos limitarnos a esta por ahora. En cualquier caso, la función de costo que utilicemos no altera el problema general.

Una posibilidad para encontrar el mínimo es hacerlo de manera analítica, utilizando las reglas de cálculo diferencial que ya conocemos, para así encontrar los puntos extremos. Sin embargo, ya vimos que la cantidad de parámetros de la red puede llegar a ser muy grande, y utilizar este método se volvería demasiado complicado o prácticamente imposible. Por suerte contamos con otros métodos.

Recordemos que cuando tenemos una función de múltiples variables su gradiente apunta en la dirección de máximo crecimiento, lo cual a su vez implica que el gradiente negativo apunta en la dirección de máximo decrecimiento. Lo que podemos hacer entonces es tomar una aproximación inicial a los parámetros θ_0 y calcular el gradiente en ese punto. Luego nos desplazamos desde θ_0 a un nuevo punto θ_1, yendo en la dirección opuesta al gradiente, con lo cual lograríamos disminuir el valor de la función. La fórmula sería

$$\theta_1 = \theta_0 - \nabla f(\theta_0). \tag{1.21}$$

Sin embargo, el gradiente es una aproximación local del comportamiento de f, y si nos desplazamos demasiado lejos en esa dirección es posible que el valor de la función ya no decrezca, sino que por el contrario, incremente. Por lo tanto, necesitamos un parámetro que regule cuanto nos desplazamos. Este parámetro se denomina *tasa de aprendizaje* y se suele denotar con η. La nueva expresión es entonces

$$\theta_1 = \theta_0 - \eta \nabla f(\theta_0). \tag{1.22}$$

El procedimiento es iterativo y una vez que tenemos el nuevo punto lo que hacemos es volver a calcular el gradiente y descender en la nueva dirección, para así conseguir un punto que esté un poco más cerca del mínimo. Obtenemos así la siguiente regla para actualizar los parámetros:

$$\theta_{k+1} = \theta_k - \eta \nabla f(\theta_k). \tag{1.23}$$

Este algoritmo se llama *descenso por gradiente*, por motivos obvios.

Volviendo al parámetro η, la razón por la cual se denomina taza de aprendizaje es que determina la velocidad con la que nos desplazamos por el espacio de parámetros, y por lo tanto cuan rápido nos acercamos al mínimo. A su vez, mientras más rápido nos acerquemos al mínimo más rápido estará aprendiendo la red a realizar su tarea. Sin embargo, como ya mencionamos antes, valores muy altos del parámetro η pueden tener el efecto contrario, llevando a que eventualmente el algoritmo pase de largo de manera reiterada el mínimo, impidiendo el aprendizaje. A veces aprender requiere ir más lento y ser más cuidadosos. Lo mismo ocurre con la red, y en ese caso es conveniente ajustar la tasa de aprendizaje para conseguir que la red no pase por alto los detalles. En la figura 5 se puede ver el efecto de diferentes tasas de aprendizaje durante el entrenamiento.

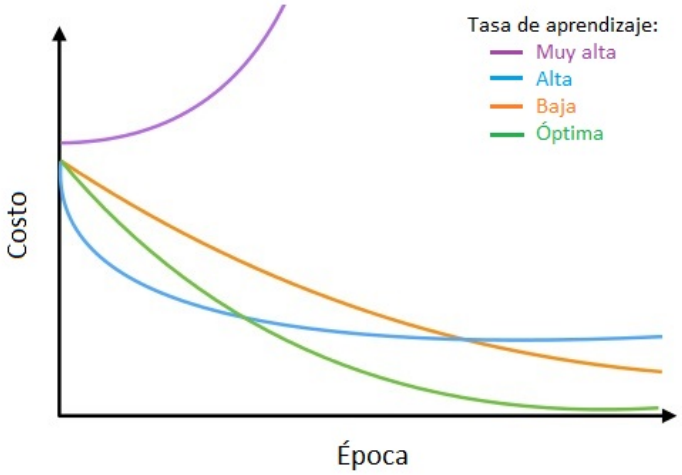


Figura 5: Efecto de diferentes tasas de aprendizaje durante el entrenamiento. Una tasa de aprendizaje demasiado alta puede ocasionar que el algoritmo de entrenamiento no pueda disminuir el costo de entrenamiento más allá de cierto punto, o en el peor de los casos puede ocasionar que el mismo aumente. Una tasa de aprendizaje demasiado baja puede hacer que el costo disminuya muy lentamente, ralentizando el aprendizaje. La tasa de aprendizaje ideal es un compromiso entre lograr una disminución significativa del costo y mantener una buena velocidad de aprendizaje.

Notemos que η es un parámetro que a diferencia de θ no es ajustado por el algoritmo de optimización, sino que debe ser configurado por separado. A este tipo de parámetros se los denomina *hiperparámetros*, y requieren de cierto cuidado en su elección para garantizar el buen rendimiento de la red neuronal. Otros hiperparámetros son, por ejemplo, el número máximo de iteraciones durante el entrenamiento, o el número de neuronas por capa. Más adelante veremos maneras en las que se pueden elegir estos hiperparámetros para optimizar el rendimiento.

Un problema que surge con el algoritmo de descenso por gradiente en el contexto en el que queremos aplicarlo es que el conjunto de entrenamiento suele consistir de miles o millones de elementos. Teniendo en cuenta que la expresión para el gradiente es

$$\nabla C = \frac{1}{n} \sum_x \nabla C_x, \tag{1.24}$$

se hace evidente que calcular cada uno de los gradientes individuales para millones de elementos puede llevar demasiado tiempo. Una alternativa a esto es un algoritmo que se conoce como *descenso por gradiente estocástico*, o SGD, por sus siglas en inglés. La idea detrás de este algoritmo es que, de acuerdo a (1.24), el gradiente no es más que un promedio, y uno puede aproximar el promedio sobre todo el conjunto tomando una muestra más pequeña, lo cual será más rápido y nos permitirá actualizar los parámetros

con mayor frecuencia. Esto se puede repetir, eligiendo en cada ocasión un subconjunto aleatorio del conjunto de entrenamiento y actualizando los parámetros de acuerdo a la formula para descenso por gradiente. Dado que el conjunto es finito, la práctica habitual consiste en mezclarlo y tomar lotes sin reposición hasta agotarlo, momento en el cual se completa una *época*. Luego se vuelve a mezclar el conjunto y se repite el proceso por tantas épocas como sea necesario.

Notemos que todavía necesitamos alguna manera de calcular ∇C_x. El enfoque simbólico no es conveniente en este caso porque a medida que la profundidad y complejidad de la red aumenta los cálculos y la expresión resultante pueden volverse poco prácticos. La alternativa más obvia en este caso sería calcular el gradiente de manera numérica. Sin embargo, eso nos implicaría el tener que aproximar cada una de las derivadas parciales utilizando, por ejemplo, la fórmula

$$\frac{df}{dw_j} \approx \frac{f(w + te_j) - f(w)}{t}. \tag{1.25}$$

Esta expresión requiere evaluar $f(w + te_j)$ para cada uno de los parámetros w_j. Si trabajamos con millones de parámetros, lo cual no es raro en este tipo de redes, lo anterior no resulta muy eficiente. La clave está en observar que la red se encuentra formada por la composición de varias capas, y esas capas a su vez se calculan componiendo ciertas operaciones. El resultado es un árbol computacional en el que la información fluye de una operación a la siguiente, y de una capa a la otra, cada operación dependiendo del resultado de la anterior. Dado que la red no es más que una composición de operaciones, para calcular el gradiente se puede aplicar la regla de la cadena, y existe una forma eficiente de realizar esto en redes como el perceptrón multicapa.

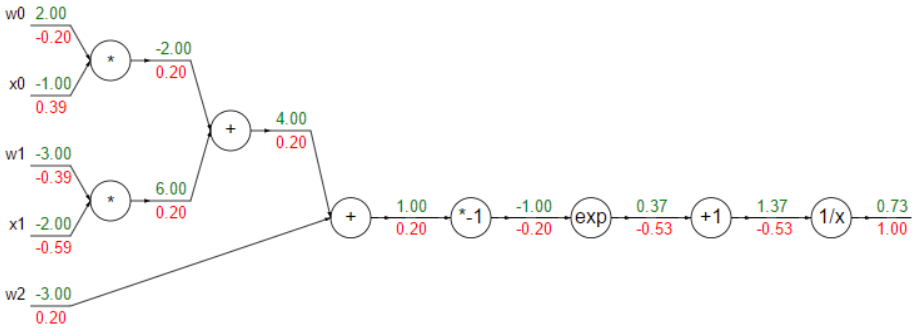


Figura 6: Gráfico computacional para una neurona 2D con activación sigmoide, donde $x = (x_1, x_2)$ y $w = (w_1, w_2, w_3)$ denotan la entrada y el vector de pesos respectivamente. Se anotaron en verde los valores intermedios durante la propagación hacia adelante y con rojo el valor de las derivadas durante la propagación hacia atrás.

Supongamos que tenemos una red formada por dos capas, $f = f^2 \circ f^1$, y denotamos las salidas de cada capa por $a^1 = f^1(x)$ y $a^2 = f^2(a^1) = f(x)$. Recordemos que cada capa

tiene sus propios parámetros, θ^2 y θ^1, que no son utilizados en las demás capas. Lo que nos va a interesar entonces, dada una función costo C, es calcular $\nabla_\theta C_x$, donde $C_x = C(f(x))$ para una entrada individual x. Para empezar, vamos a denotar las derivadas parciales respecto a las activaciones como

$$\delta^2 = \frac{\partial C_x}{\partial a^2}, \quad \delta^1 = \frac{\partial C_x}{\partial a^1}. \tag{1.26}$$

La derivada respecto a θ^2 es

$$\frac{\partial C_x}{\partial \theta^2} = \frac{\partial C_x}{\partial a^2}\frac{\partial a^2}{\partial \theta^2} = \delta^2 \frac{\partial a^2}{\partial \theta^2}. \tag{1.27}$$

Por otra parte, si calculamos las derivadas respecto a las activaciones de la primer capa, resulta

$$\delta^1 = \frac{\partial C_x}{\partial a^1} = \frac{\partial C_x}{\partial a^2}\frac{\partial a^2}{\partial a^1} = \delta^2 \frac{\partial a^2}{\partial a^1}; \tag{1.28}$$

y la derivada respecto a θ^1 es

$$\frac{\partial C_x}{\partial \theta^1} = \frac{\partial C_x}{\partial a^2}\frac{\partial a^2}{\partial a^1}\frac{\partial a^1}{\partial \theta^1} = \delta^1 \frac{\partial a^1}{\partial \theta^1}. \tag{1.29}$$

Si examinamos las ecuaciones vemos que para $k = 1, 2$, δ^k se calcula a partir de información proveniente de la capa $k+1$, mientras que las derivadas respecto a θ^k se calculan multiplicando por $\partial a^k / \partial \theta^k$, que depende de la capa k. En otras palabras, para calcular $\partial C_x / \partial \theta^k$ podemos ir propagando hacia atrás la información, calculando los sucesivos δ^k de manera recursiva. Una vez obtenido δ^k para cada k, se puede calcular $\partial C_x / \partial \theta^k$ para cada capa, con lo cual uno obtiene $\partial C_x / \partial \theta$. El resultado es que si uno guarda un registro de las activaciones de cada una de las capas, lo cual se puede hacer al evaluar la red en x, entonces ya no es necesario propagar la información reiteradas veces a través de la red para calcular las derivadas del costo respecto a cada parámetro, sino que con dos pasadas alcanzan: una vez hacia adelante para obtener la salida y las activaciones de cada capa, y una vez hacia atrás para ir obteniendo los gradientes. Si estamos trabajando con millones de parámetros esto supone un ahorro importante en términos de eficiencia. Este método se conoce como *backpropagation* y se verá en detalle su implementación para MLP en la sección 1.7.

1.6. Clasificación de dígitos con redes neuronales

Un problema concreto que nos va a interesar atacar en este trabajo es el de implementar una red neuronal capaz de clasificar dígitos escritos a mano. Lo primero que necesitamos es un conjunto de entrenamiento con el cual entrenar la red para realizar dicha tarea, y existe uno muy popular para este tipo de clasificador, que recibe el nombre de MNIST.

MNIST es una base de datos de dígitos escritos a mano que se divide en un conjunto de entrenamiento conteniendo 60000 imágenes y uno de prueba de 10000 imágenes. Esta base de datos es un subconjunto de una más grande, proveniente de NIST (National Institute of Standards and Technology); para ser precisos, MNIST se construyó a partir de la NIST

Special Database 3 y Special Database 1, que contienen imágenes binarias de dígitos escritos a mano. Originalmente SD-3 fue designado como conjunto de entrenamiento y SD-1 como conjunto de prueba, pero resulta que SD-3 es más claro y fácil de reconocer que SD-1; la razón de esto se encuentra en que SD-3 fue tomado de empleados de la oficina de censos, mientras que SD-1 fue tomado de estudiantes de secundario. Esta discrepancia hace que los resultados no sean independientes de cuales conjuntos se están usando como entrenamiento y como prueba respectivamente. Para solucionar esto lo que se hizo fue mezclar SD-1 y SD-3, tomando 30000 muestras de cada uno para el conjunto de entrenamiento y otras 5000 de cada uno para el conjunto de prueba. También se tomó la precaución de que los autores de los dígitos en el conjunto de entrenamiento fueran distintos a los del conjunto de prueba. Las imágenes originales fueron además normalizadas para encajar en una caja de 20×20 preservando su razón de aspecto y luego centradas en una imagen de 28×28 por medio de su centro de masa. Como resultado del algoritmo de normalización a las imágenes también se les aplicó antialiasing, por lo cual ya no son binarias, sino que se encuentran en escala de grises. Es debido a estas modificaciones que el conjunto recibe el nombre de MNIST, que corresponde a "Modified NIST".

La razón por la cual este conjunto es popular es que la clasificación de dígitos es un problema relativamente sencillo que posee muchas de las características esenciales de otros más complejos, como la clasificación de imágenes, del cual no es más que un caso particular. Esto hace que MNIST sea un buen conjunto para desarrollar las técnicas básicas que se usan en redes neuronales, como también para la experimentación.

1.7. Backpropagation para MLP

En esta sección vamos a describir con más detalle lo que se introdujo antes sobre backpropagation, ya que lo necesitaremos para implementar nuestro clasificador.

El objetivo del algoritmo de backpropagation es calcular la derivada parcial $\partial C/\partial\theta$, que luego usaremos en el algoritmo de optimización para actualizar los parámetros θ. Para poder usar este algoritmo hay dos hipótesis que la función costo debe cumplir. La primera es que la función costo C se pueda escribir como un promedio de los costos individuales C_x, y la segunda es que el costo se pueda escribir como una función de las salidas de la red.

La razón por la que requerimos la primer condición es que el algoritmo de backpropagation lo que nos va a permitir es calcular la derivada parcial $\partial C_x/\partial\theta$ para cada ejemplo individual del conjunto de entrenamiento, mientras que la segunda condición la requerimos ya que el algoritmo parte de la salida y propaga la información hacia atrás para obtener los gradientes, como se verá en breve.

Para calcular los gradientes vamos a necesitar un intermediario, al que denotaremos con δ^l y que definimos como

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}, \tag{1.30}$$

donde $z_j^l = w_{j:}^l \cdot x + b_j^l$, y $l \in \{1, \ldots, L\}$ denota la capa en la que nos encontramos. Observemos que si realizamos un pequeño cambio Δz_j, esto tendrá un impacto en el costo, que se puede medir por medio de $(\partial C/\partial z_j^l)\Delta z_j^l$. Por lo tanto, si $\partial C/\partial z_j^l$ es una cantidad grande, entonces podemos disminuir mucho el costo eligiendo Δz_j^l con signo

contrario a $\partial C/\partial z_j^l$, mientras que si $\partial C/\partial z_j^l$ es muy pequeño, entonces no es posible mejorar mucho más el costo modificando z_j^l. En ese último caso se podría decir que hasta donde sabemos, el costo ya está bastante cerca del óptimo para esa neurona. Es en este sentido que a la cantidad δ^l se la conoce como *error*, ya que se puede pensar como una medida de lo que le falta aprender a la red. Lo que vamos a lograr con backpropagation (o backprop, para hacerlo más corto), es calcular los errores δ^l para cada capa l, lo cual luego relacionaremos con $\nabla_\theta C$, que es lo que realmente nos interesa.

El algoritmo se basa en cuatro ecuaciones. Empezamos calculando el error para la capa más externa, que es la de salida, a la que denotaremos con L.

$$\delta_j^L = \frac{\partial C}{\partial z_j^L} = \sum_k \frac{\partial C}{\partial a_k^L}\frac{\partial a_k^L}{\partial z_j^L} = \sum_k \frac{\partial C}{\partial a_k^L}\frac{\partial \sigma(z_k^L)}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L). \tag{1.31}$$

Le sigue la ecuación que nos permite propagar el error a través de las capas, o en otros términos, calcular δ^l a partir de δ^{l+1}:

$$\begin{aligned}
\delta_j^l = \frac{\partial C}{\partial z_j^l} &= \sum_k \frac{\partial C}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} \\
&= \sum_{k,i} \delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial a_i^l}\frac{\partial a_i^l}{\partial z_j^l} \\
&= \sum_{k,i} \delta_k^{l+1}\frac{\partial (w_{k:}^{l+1}\cdot a^l + b_k^{l+1})}{\partial a_i^l}\frac{\partial \sigma(z_i^l)}{\partial z_j^l} \\
&= \sum_k \delta_k^{l+1}\frac{\partial (\sum_n w_{kn}^{l+1} a_n^l + b_k^{l+1})}{\partial a_j^l}\sigma'(z_j^l) \\
&= \sum_k \delta_k^{l+1} w_{kj}^{l+1}\sigma'(z_j^l).
\end{aligned} \tag{1.32}$$

Ahora necesitamos derivar las ecuaciones que permiten relacionar δ con los gradientes respecto a w.

$$\begin{aligned}
\frac{\partial C}{\partial w_{ij}^l} &= \sum_k \frac{\partial C}{\partial z_k^l}\frac{\partial z_k^l}{\partial w_{ij}^l} \\
&= \sum_k \delta_k^l \frac{\partial (w_{k:}^l \cdot a^{l-1} + b_k^l)}{\partial w_{ij}^l} \\
&= \sum_{k,n} \delta_k^l \frac{\partial (w_{kn}^l a_n^{l-1} + b_k^l)}{\partial w_{ij}^l} \\
&= \delta_i^l a_j^{l-1}.
\end{aligned} \tag{1.33}$$

De manera análoga para b:

$$\frac{\partial C}{\partial b_j^l} = \sum_k \frac{\partial C}{\partial z_k^l}\frac{\partial z_k^l}{\partial b_j^l} = \sum_k \delta_k^l \frac{\partial (w_{k:}^l \cdot a^{l-1} + b_k^l)}{\partial b_j^l} = \delta_j^l. \tag{1.34}$$

Los resultados anteriores se pueden expresar de forma más compacta como:

$$\delta^L = \nabla_a C \odot \sigma'(z^L), \tag{1.35}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l), \tag{1.36}$$

$$\frac{\partial C}{\partial w^l} = \delta^l (a^{l-1})^T, \tag{1.37}$$

$$\frac{\partial C}{\partial b^l} = \delta^l. \tag{1.38}$$

Si observamos la ecuación para $\partial C/\partial w^l$ vemos que si las activaciones a^{l-1} son pequeñas, entonces el gradiente respecto a los pesos w^l va a tender a ser pequeño, con lo cual las neuronas en esa capa van a aprender más lentamente durante el entrenamiento. Lo mismo ocurre si δ^l es pequeño.

De manera similar, de la ecuación para la capa de salida se puede ver que si $\sigma'(z_j^L) \approx 0$ entonces esa neurona va a aprender más lentamente. Usando una activación sigmoide, esto ocurre cada vez que $z_j^L \approx 1$ o $z_j^L \approx 0$, que es lo mismo que decir que las neuronas de la capa de salida aprenden más lentamente cuando presentan activaciones muy altas o muy bajas. Este fenómeno se conoce como saturación, y es algo que se tiene en cuenta al diseñar funciones de activación con mejores rendimientos, como ya se verá más adelante.

Con las ecuaciones obtenidas podemos construir un algoritmo para calcular el gradiente $\partial C/\partial w$ de la siguiente manera:

1. Entrada: inicializar la primer capa de activaciones por medio de $a^1 = x$, donde x es la entrada.

2. Feedforward: calcular $z^l = w^l a^{l-1} + b^l$ para $l = 2, \ldots, L$.

3. Error de salida: calcular $\delta^L = \nabla_a C \odot \sigma'(z^L)$.

4. Propagación hacia atrás del error: calcular $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$, para $l = L-1, \ldots, 2$.

5. Salida: los gradientes de la función costo respecto a los parámetros w y b están dados por
$$\frac{\partial C}{\partial w^l} = \delta^l (a^{l-1})^T, \quad \frac{\partial C}{\partial b^l} = \delta^l.$$

1.8. Implementación del MLP

El lenguaje que usaremos en la implementación será Python. Python es un lenguaje interpretado de alto nivel y de propósito general que puede ser extendido para el uso científico y el análisis de datos mediante bibliotecas como Numpy y Scipy, las cuales nos proveen de un entorno para realizar calculo numérico y funcionalidades comúnmente usadas en ingeniería y ciencia. Este lenguaje también es frecuentemente usado en machine learning y deep learning, a veces descripto como el estándar de facto para estas disciplinas, y existen frameworks especializados para redes neuronales, como PyTorch o TensorFlow, los cuales permiten no solo agilizar el desarrollo, sino que brindan dos conveniencias muy importantes: por un lado la capacidad de utilizar la GPU para paralelizar los cálculos y acelerar el entrenamiento; por otro, implementaciones más generales del algoritmo de

backpropagation, las cuales nos proporcionan de manera automática los gradientes de la red a partir de su gráfico computacional. En lo que sigue vamos a limitarnos a una implementación básica realizada completamente en Numpy, pero más adelante veremos implementaciones basadas en los frameworks antes mencionados.

Comenzamos importando las bibliotecas que vamos a usar y definiendo ciertos parámetros.

```
import numpy as np
import numpy.random as rnd

neurons = [784, 400, 100, 10]
L = len(neurons)
w, b, dw, db, a, z = [L*[0] for k in range(6)]
```

Las primeras dos líneas importan la biblioteca de Numpy, a la que llamaremos `np` por brevedad, y un submódulo para generar números aleatorios, al que denotamos por `rnd`. La variable `neurons` define el tamaño de las capas y no es más que una lista conteniendo el número de neuronas por capa. Por último creamos las variables que contendrán los pesos, los sesgos, los gradientes, las activaciones y los estados de las capas. Estas variables consisten por el momento de listas inicializadas con tantos ceros como número de capas en la red. Los ceros no son más que marcadores de posición que luego serán reemplazados con matrices conteniendo los valores reales. La idea es que si la expresión w^k denota la matriz de pesos de la capa k, el equivalente en nuestro código será escribir `w[k]`. De manera análoga con las demás variables.

Ahora vamos a definir la función de inicialización para los parámetros.

```
def initialize():
    for k in range(1, L):
        in_size = neurons[k-1]
        out_size = neurons[k]
        w[k] = rnd.randn(out_size, in_size)
        b[k] = rnd.randn(out_size, 1)
```

Lo que esta función hace es inicializar los pesos y los sesgos de cada capa con matrices generadas de manera aleatoria a partir de una distribución normal estándar. Para cada capa el tamaño de la entrada será igual al número de neuronas de la capa anterior, mientras que la salida es igual al número de neuronas de la capa actual. A estos valores los llamamos `in_size` y `out_size`. Usando esta información construimos las matrices de pesos, de dimensión `out_size` $\times$ `in_size`, y los vectores de sesgos, de dimensión `out_size` $\times 1$. Nótese que `w[0]` y `b[0]` se dejan sin inicializar. La razón de esto es que la capa de entrada no es una capa compuesta de neuronas y por lo tanto no le corresponden ni pesos ni sesgos.

El próximo paso es escrbir la rutina de `feedforward`, que no es más que una descripción en Python de la ecuación (1.18), con la cual pasamos la información a través de la red para obtener una salida.

```
def feedforward(x):
    a[0] = x
```

```
    z[0] = x
    for k in range(1, L):
        z[k] = w[k] @ a[k-1] + b[k]
        a[k] = sigmoid(z[k])
    return a[-1]
```

Las primeras dos lineas inicializan la activación y estado de la capa de entrada. A partir de ahí vamos calculando para las sucesivas capas las cantidades $z^k = w^k a^{k-1} + b^k$ y $a^k = \sigma(z^k)$. La última activación es la que contiene la salida de la red, que corresponde en nuestro código a `a[L-1]`.

Ahora vamos a escribir el algoritmo de `backprop`, tal como aparece en la sección 1.7.

```
def backprop(x, y):
    feedforward(x)
    delta = (a[-1] - y) * sigmoid_prime(z[-1])
    dw[-1] = delta @ a[-2].T
    db[-1] = np.sum(delta, axis=1, keepdims=True)
    for k in reversed(range(1, L-1)):
        delta = (w[k+1].T @ delta) * sigmoid_prime(z[k])
        dw[k] = delta @ a[k-1].T
        db[k] = np.sum(delta, axis=1, keepdims=True)
```

El paso 1 y 2 del algoritmo es realizado al ejecutar la función `feedforward`. Luego le sigue el paso 3, en la siguiente línea, donde calculamos $\delta^l = \nabla_a C \odot \sigma'(z^L)$. La notación `a[-k]` nos permite atravesar las listas en orden inverso, por lo que `a[-1]` es la activación de la última capa, y de manera análoga con las demás variables. Los gradientes de los pesos y los sesgos los iremos calculando sobre la marcha, de modo que una vez obtenido δ^L calculamos $\partial C/\partial w^L$ y $\partial C/\partial b^L$. Al terminar con la capa de salida entramos en el bucle, donde iremos propagando hacia atrás el error por el resto de las capas, de acuerdo a la formula (1.36), y actualizando los gradientes utilizando las ecuaciones (1.37) y (1.38).

Le sigue ahora la implementación del algoritmo de SGD, responsable del entrenamiento de la red.

```
def SGD(trng_set, val_set, loss_fn, epochs=30, bat_size=100, eta=1):
    x, y = trng_set
    trng_size = x.shape[1]
    for n in range(epochs):
        shf_idx = rnd.permutation(trng_size)
        x, y = x[:, shf_idx], y[:, shf_idx]
        for k in range(0, trng_size, bat_size):
            backprop(x[:, k:k+bat_size], y[:, k:k+bat_size])
            for k in range(1, L):
                w[k] =  w[k] - eta/bat_size * dw[k]
                b[k] =  b[k] - eta/bat_size * db[k]
        acc, loss = performance(val_set, loss_fn)
        print('epoch: %2d | acc: %2.2f%% | loss: %.5f' % (n+1, acc*100, loss))
```

Esta función recibe como parámetros el número de épocas, el tamaño del minilote y la tasa de entrenamiento. Por cada época lo que hacemos es permutar el conjunto de

entrenamiento y luego extraemos de manera sucesiva minilotes de tamaño `bat_size`. Sobre estos lotes aplicamos backprop para obtener los gradientes, los cuales utilizamos luego para actualizar las matrices de pesos y sesgos de acuerdo al algoritmo de descenso por gradiente. Una vez agotado el conjunto de entrenamiento mostramos la exactitud y costo actuales de la red y procedemos a empezar una nueva época.

En la rutina anterior hemos usado una función que nos permite monitorear el progreso del entrenamiento, la cual se encuentra definida a continuación:

```
def performance(test_set, loss_fn):
    x, y = test_set
    test_size = x.shape[1]
    feedforward(x)
    pred = np.argmax(a[-1], axis=0)
    label = np.argmax(y, axis=0)
    acc = np.mean(pred == label)
    loss = loss_fn(a[-1], y) / test_size
    return acc, loss
```

Esta función evalúa la red sobre el conjunto de prueba y luego compara la predicción que hace para cada entrada con el rótulo que realmente le corresponde. Esto nos da un vector que contiene unos para cada elemento clasificado correctamente y ceros para los que no. Calculando la media sobre este vector obtenemos la exactitud como un número entre 0 y 1. Eso es lo que hacemos para obtener el valor de `acc`. También calculamos el costo o pérdida total para la red y se lo asignamos a `loss`, para lo cual es necesario pasar como argumento la función de costo individual que estamos utilizando. Finalmente regresamos ambos valores.

Las definiciones de la función sigmoide y su derivada, como también las del costo cuadrático y su gradiente, son copia fiel de sus respectivas ecuaciones:

```
def quad_cost(a, y):
    return 1/2 * np.sum((a - y)**2)

def quad_grad(a, y):
    return a - y

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_prime(x):
    s = sigmoid(x)
    return s * (1 - s)
```

Por último, necesitamos una función para cargar los datos:

```
#%% Carga de datos
def load_data(trng_size=50000, val_size=10000,
              shuffle=False, normalize=True):
    total_size = trng_size + val_size
    x, y = np.load('../data/trng_data.npy')
```

```
    if shuffle:
        shf_idx = np.random.permutation(x.shape[1])
        x, y = x[:, shf_idx], y[:, shf_idx]
    if normalize:
        x = (x - np.mean(x, axis=0))/np.std(x, axis=0)
    trng_set = (x[:,0:trng_size], y[:,0:trng_size])
    val_set = (x[:,trng_size:total_size], y[:,trng_size:total_size])
    return trng_set, val_set
```

Como opción nos permite elegir el tamaño del conjunto de entrenamiento y el tamaño del conjunto que usamos para verificar el rendimiento, que por defecto será de 50000 y 10000 respectivamente. También podemos elegir si queremos que los datos sean normalizados y si queremos mezclarlos. Lo habitual es que el conjunto sea normalizado, ya que si no lo hacemos el rendimiento de la red puede verse afectado de manera negativa.

Ahora solo queda correr el programa. Para eso lo que tenemos que hacer es cargar los datos, inicializar la red y entrenar usando SGD. Entrenaremos por 30 épocas con un minilote de tamaño 32 y tasa de entrenamiento igual a 3.

```
trng_set, val_set = load_data()
initialize()
SGD(trng_set, val_set, quad_cost, epochs=30, bat_size=32, eta=3)
```

El resultado del entrenamiento es el siguiente:

```
epoch:  1 | acc: 56.78% | loss: 0.25440
epoch:  2 | acc: 66.69% | loss: 0.20188
epoch:  3 | acc: 75.69% | loss: 0.15553
epoch:  4 | acc: 76.16% | loss: 0.14973
epoch:  5 | acc: 76.55% | loss: 0.14613
...
epoch: 26 | acc: 94.15% | loss: 0.05069
epoch: 27 | acc: 94.11% | loss: 0.05105
epoch: 28 | acc: 94.18% | loss: 0.05103
epoch: 29 | acc: 94.13% | loss: 0.05080
epoch: 30 | acc: 94.20% | loss: 0.05063
```

Podemos ver que para la época 30 la red llegó a un 94.20 % de exactitud. Para una primera aproximación los resultados son aceptables. No obstante, cabe destacar que los resultados no son consistentes entre ejecuciones. En una segunda ronda del programa el entrenamiento arrojó los siguientes resultados:

```
epoch:  1 | acc: 49.61% | loss: 0.28646
epoch:  2 | acc: 49.62% | loss: 0.27855
epoch:  3 | acc: 50.74% | loss: 0.27211
epoch:  4 | acc: 65.33% | loss: 0.20831
epoch:  5 | acc: 66.74% | loss: 0.19559
...
epoch: 26 | acc: 76.79% | loss: 0.13755
```

```
epoch: 27 | acc: 76.73% | loss: 0.13750
epoch: 28 | acc: 76.73% | loss: 0.13751
epoch: 29 | acc: 76.70% | loss: 0.13791
epoch: 30 | acc: 76.73% | loss: 0.13760
```

El motivo de esto es que la matriz de parámetros es inicializada de manera diferente en cada ejecución. Recordemos que la inicialización de los parámetros w y b determina el punto inicial a partir del cual comenzamos el proceso de descenso por gradiente estocástico. En el primer caso la inicialización fue más favorable y el algoritmo pudo acercarse al mínimo más fácilmente, mientras que en el segundo caso la inicialización no ayudó y ya no pudo mejorar la exactitud por encima del 76 %. Esto nos da la pauta de que una de las primeras cosas que podemos considerar para aumentar el rendimiento de la red y lograr resultados más consistentes es elegir una mejor inicialización. Sin embargo eso no es todo.

Además de la inicialización, la elección de la función de activación utilizada en las neuronas de la red también influye durante el entrenamiento y se pueden construir activaciones con mejores propiedades que la sigmoide. La función sigmoide tiene tendencia a un fenómeno conocido como saturación, que ocasiona que los gradientes propagados durante el proceso de backprop sean pequeños, ralentizando el aprendizaje. Ciertas activaciones se construyen para evitar este problema.

Otra manera de mejorar la exactitud de la red es la introducción de lo que se conoce como métodos de regularización, cuyo objetivo es mejorar la capacidad de la red para generalizar a datos que no vio durante el entrenamiento, siendo esto lo que en el fondo nos interesa. Nosotros no entrenamos la red solo con el objetivo de que pueda clasificar correctamente el conjunto de entrenamiento, sino que esperamos que la red sea capaz de utilizar esta experiencia para clasificar de manera aceptable nuevas entradas que se le presenten. La regularización consiste de técnicas que procuran evitar que la red aprenda detalles específicos del conjunto de entrenamiento que no le ayudan al momento de generalizar a otros datos. Se sacrifica de esta manera la exactitud sobre el conjunto de entrenamiento a fin de mejorar el rendimiento sobre entradas que la red no observó previamente.

Por otra parte, dado que el entrenamiento se reduce a un problema de optimización, esto nos lleva a examinar la función costo que buscamos optimizar y el método utilizado para obtener la solución. Es posible elegir funciones costo con mejores propiedades que el costo cuadrático que hemos usado hasta ahora. También existen refinamientos sobre el algoritmo de gradiente estocástico que con pocas modificaciones consiguen rendimientos superiores durante el entrenamiento.

El objetivo de las secciones siguientes será explorar las distintas maneras de mejorar los resultados obtenidos hasta el momento.

2. Mejorando el Aprendizaje de la Red Neuronal

2.1. Función costo. Entropía cruzada

Del error se aprende y muchas veces es más fácil para nosotros aprender cuando estos son contundentes. Los errores graves son más obvios y pueden sugerir cambios claros en el comportamiento a fin de evitar que se vuelvan a repetir, mientras que no siempre queda muy claro como mejorar si el error es sutil. De manera análoga, nosotros esperamos que la red ajuste rápidamente su comportamiento frente a la presencia de errores muy evidentes al momento de clasificar.

Para ver si eso es lo que ocurre con nuestra implementación actual, lo que haremos es armar una red muy sencilla que deberá aprender a mapear el vector $x = (0, 1)$ al vector $y = (1, 0)$. La red contará solo con una capa de entrada y una de salida, por lo que podemos describirla meramente como la función

$$f(x) = \sigma(wx + b), \tag{2.1}$$

donde w es una matriz de 2×2. En principio vamos a hacer

$$w = \begin{bmatrix} 2 & -3 \\ -1 & 4 \end{bmatrix}, \qquad b = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, \tag{2.2}$$

que no es más que una elección arbitraria para ver qué ocurre con la red. Obtenemos así que la salida inicial de la red es $f(x) = (0.119; 0.881)$. Si ahora entrenamos la red y monitoreamos la evolución del costo a lo largo de las épocas de entrenamiento obtenemos el gráfico de la figura 7. Del gráfico se puede ver que la red aprende rápidamente los parámetros que le permiten disminuir el costo, tal como esperamos que ocurra.

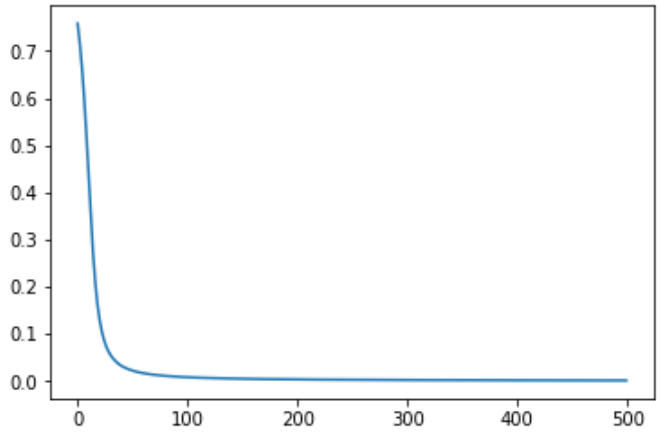


Figura 7: Evolución del costo cuadrático por época.

Ahora vamos a elegir

$$w = \begin{bmatrix} 0 & 0 \\ 0 & 5 \end{bmatrix}, \qquad b = \begin{bmatrix} -5 \\ 0 \end{bmatrix}. \tag{2.3}$$

Con estos valores, la red regresa como valor inicial el vector $(0.007; 0.993)$, que claramente está en el extremo opuesto del valor que debe aprender. Dado que el error de clasificación es evidente, nosotros esperaríamos que la red corrija rápidamente los valores de los

parámetros durante el entrenamiento; no obstante, la gráfica de la evolución del costo es la que aparece en la figura 8.

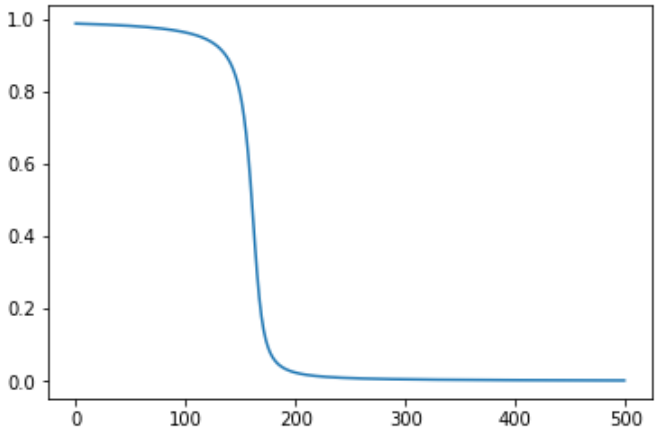


Figura 8: Evolución del costo cuadrático por época. En esta ocasión el valor inicial de la red se encuentra en el extremo opuesto del que debe aprender. Notesé que el aprendizaje es lento al principio.

Se puede ver que la red no aprende rápidamente al principio, sino que por el contrario, aprende muy lentamente y recién cerca de la época 150 realmente empieza a disminuir el costo. Este comportamiento es el opuesto del que esperamos y sugiere la existencia de limitaciones con algún componente de nuestra implementación actual. Notemos además que esto no está relacionado con la elección del valor de η, sino que es un fenómeno cualitativo de la red. Si eligiéramos un valor más alto para η podríamos aumentar la velocidad de aprendizaje, pero eso no cambiaría el hecho de que la red tiene un aprendizaje más lento al principio que en etapas tardías. El valor de η utilizado en el gráfico anterior fue 0.5. Si ahora utilizamos un valor de 2, el resultado es una evolución del costo que tiene esencialmente el mismo comportamiento, como se ve en la figura 9, y es ese comportamiento lo que nos interesaría corregir.

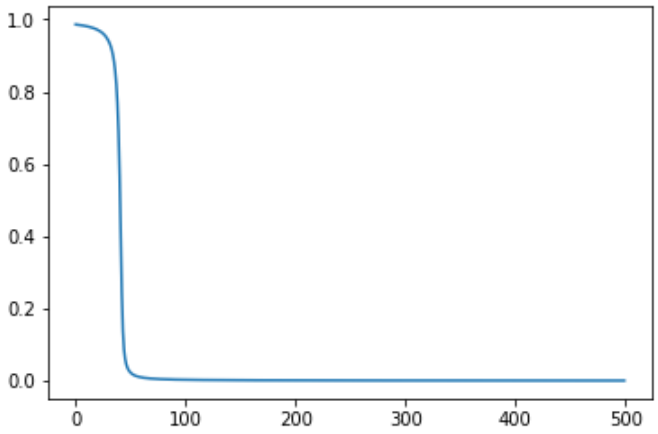


Figura 9: Se modificó la tasa de aprendizaje para acelerar el entrenamiento, sin embargo, en términos cualitativos, el comportamiento sigue siendo el mismo.

Recordemos que las neuronas de nuestra red aprenden los valores de sus parámentros utilizando la fórmula

$$\theta_{k+1} = \theta_k - \eta \nabla C(\theta_k), \tag{2.4}$$

por lo que la velocidad de aprendizaje no solo está determinada por el hiperparámetro η, sino también por el gradiente $\nabla_\theta C$, de modo que si el aprendizaje es lento al principio puede ser porque ese gradiente es pequeño. Si combinamos las ecuaciones (1.35) y (1.37) usadas en backprop vemos que la derivada del costo respecto a los pesos para nuestra red de una sola capa con costo cuadrático es

$$\frac{\partial C}{\partial w} = \left((a-y) \odot \sigma'(z)\right)\, x^T. \tag{2.5}$$

Observando esta ecuación tenemos una pista de lo que origina nuestro problema. La función de activación σ, que en nuestro caso es la sigmoide, se vuelve cada vez más plana a medida que se acerca a los valores 0 y 1. Esto significa que tanto si la red está muy cerca de la salida correcta, como también si está muy equivocada, σ' va a tender a ser pequeño, lo cual a su vez va a generar que $\partial C/\partial w$ sea pequeño, ralentizando el aprendizaje. Esto es precisamente lo que ocurrió en el ejemplo anterior.

Una solución para nuestro problema sería encontrar una activación que no presentara el problema de saturación que vemos en la sigmoide. Sin embargo, sería aun mejor si consiguiéramos eliminar el término $\sigma'(z)$ de la ecuación anterior, ya que de esa manera la velocidad de aprendizaje dependería de $a - y$, es decir, de la diferencia entre el valor correcto y la predicción que realiza la red. Este es precisamente el comportamiento que esperamos: mientras más equivocada esté la red más fuerte será el gradiente y más rápido el aprendizaje.

Lo que necesitaremos entonces será construir una función costo C de tal manera que $\nabla_a C$ cancele de alguna manera al término σ'. Si ese fuera el caso tendríamos, para un solo ejemplo de entrenamiento x, las ecuaciones

$$\frac{\partial C}{\partial w} = (a-y)x^T \tag{2.6}$$

$$\frac{\partial C}{\partial b} = (a-y). \tag{2.7}$$

Sabemos por la ecuación (1.38) que

$$\frac{\partial C}{\partial b} = \delta = \frac{\partial C}{\partial a}\sigma'(z). \tag{2.8}$$

También sabemos que

$$\sigma'(z) = \sigma(z)(1-\sigma(z)) = a(1-a). \tag{2.9}$$

Reemplazando y despejando obtenemos

$$\frac{\partial C}{\partial a} = \frac{\partial C/\partial b}{a(1-a)} = \frac{a-y}{a(1-a)}. \tag{2.10}$$

Si ahora integramos llegamos a

$$C = -\left[y\log(a) + (1-y)\log(1-a)\right] + \text{constante}, \tag{2.11}$$

donde la constante puede ser descartada. Esta es la expresión para el costo individual. Para el costo completo debemos promediar sobre todos los ejemplos del conjunto de entrenamiento, con lo que obtenemos

$$C = -\frac{1}{n}\sum_x y_x \log(a_x) + (1 - y_x)\log(1 - a_x). \tag{2.12}$$

Esta función costo se conoce como *cross entropy* o entropía cruzada.

Si ahora utilizamos nuestra nueva función para el ejemplo que trabajamos antes, podemos ver en la figura 10 que la evolución del costo es diferente, disminuyendo rápidamente desde el principio, como esperábamos lograr.

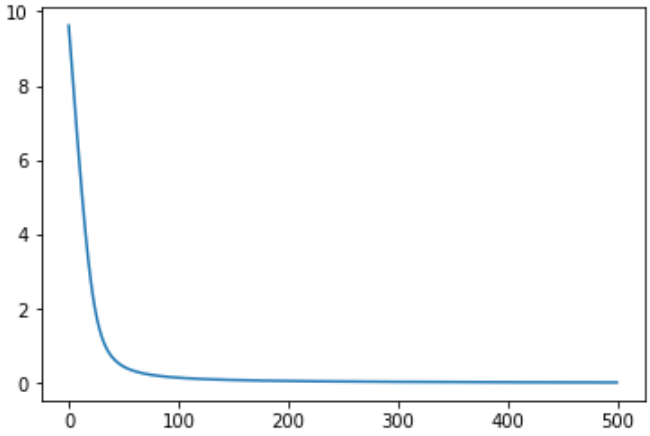


Figura 10: Evolución del costo por época, pero ahora usando la entropía cruzada como función costo.

El análisis anterior aplica a redes con más capas, siendo necesario solo un cambio notacional, ya que de acuerdo a (1.35), (1.37) y (1.38), el gradiente de la función costo solo afecta directamente a las derivadas $\partial C/\partial w^L$ y $\partial C/\partial b^L$. Basta entonces con reemplazar (x, a, w, b) por (a^{L-1}, a^L, w^L, b^L), y las ecuaciones anteriores son validas tal y como están.

La implementación de esta nueva función costo requiere pequeños cambios en el código. El primero es la modificación de la función backprop. Recordemos que la definición original era

```
    def backprop(x, y):
        feedforward(x)
        delta = (a[-1] - y) * sigmoid_prime(z[-1])
        dw[-1] = delta @ a[-2].T
        db[-1] = np.sum(delta, axis=1, keepdims=True)
        for k in reversed(range(1, L-1)):
            delta = (w[k+1].T @ delta) * sigmoid_prime(z[k])
            dw[k] = delta @ a[k-1].T
            db[k] = np.sum(delta, axis=1, keepdims=True)
```

Utilizando la nueva función costo podemos eliminar la derivada de la sigmoide, por lo que ahora pasa a ser:

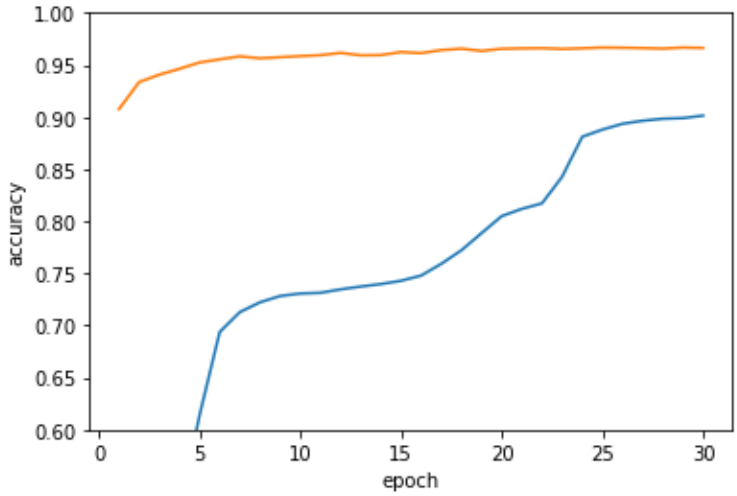


Figura 11: Comparación de la evolución de dos rondas de entrenamiento, una utilizando como función objetivo el costo cuadrático (azul) y la otra utilizando la entropía cruzada (naranja).

```
    def backprop(x, y):
        feedforward(x)
        delta = (a[-1] - y)
        dw[-1] = delta @ a[-2].T
        ...
```

El segundo cambio es la definición de la nueva función costo, que es como sigue:

```
    def cross_entropy_cost(a, y):
        return -np.sum(y*np.log(a) + (1 - y)*np.log(1 - a))
```

Con esto ya estamos listos para ejecutar nuestro nuevo programa. El resultado del entrenamiento es el siguiente:

```
epoch:  1 | acc: 90.84% | loss: 0.60206
epoch:  2 | acc: 92.91% | loss: 0.48694
epoch:  3 | acc: 93.95% | loss: 0.40654
epoch:  4 | acc: 94.87% | loss: 0.34208
epoch:  5 | acc: 95.00% | loss: 0.34142
...
epoch: 26 | acc: 96.65% | loss: 0.30740
epoch: 27 | acc: 96.70% | loss: 0.31374
epoch: 28 | acc: 96.68% | loss: 0.31333
epoch: 29 | acc: 96.64% | loss: 0.31853
epoch: 30 | acc: 96.59% | loss: 0.32261
```

Se puede observar que no solo obtuvimos mejor rendimiento para el final del entrenamiento, sino que la red aprendió notablemente más rápido que en nuestra implementación original, obteniendo una exactitud del 90.84 % en la primer época de entrenamiento.

Recordemos que con el costo cuadrático el entrenamiento logró un 56.78 % al inicio y fueron necesarias varias épocas para alcanzar una exactitud superior al 90 %.

2.2. Inicialización

Supongamos que tenemos una entrada x y una neurona lineal con pesos w inicializados de manera aleatoria. La salida z está dada por

$$z = \sum_{k}^{n} w_k x_k. \tag{2.13}$$

Lo que nos va a interesar será calcular la varianza de z. Si asumimos que los w_k y x_k son independientes entre sí, vamos a tener que

$$\operatorname{Var}[w_k x_k] = \operatorname{E}[w_k]^2 \operatorname{Var}[x_k] + \operatorname{E}[x_k]^2 \operatorname{Var}[w_k] + \operatorname{Var}[x_k] \operatorname{Var}[w_k]. \tag{2.14}$$

Ahora podemos asumir que x_k tiene media cero, ya que es usual normalizar los elementos x del conjunto de entrenamiento. En lo que respecta a los pesos w_k, basta con generarlos a partir de una distribución con media cero. En ese caso la ecuación anterior se simplifica, quedando

$$\operatorname{Var}[w_k x_k] = \operatorname{Var}[x_k] \operatorname{Var}[w_k]. \tag{2.15}$$

Si ahora asumimos que x_k y w_k son i.i.d tenemos que

$$\operatorname{Var}[z] = \sum_{k}^{n} \operatorname{Var}[w_k] \operatorname{Var}[x_k] = n \operatorname{Var}[w_i] \operatorname{Var}[x_i]. \tag{2.16}$$

Recordemos que hasta ahora hemos inicializado los pesos a partir de una distribución normal estándar, por lo que en ese caso,

$$\operatorname{Var}[z] = n \operatorname{Var}[x_i]. \tag{2.17}$$

Dado que n suele ser un número relativamente grande, podemos ver que la varianza de z también será grande, haciendo más probable que $|z| \gg 1$. Si ahora aplicamos la activación sigmoide a z vemos que $\sigma(z)$ tenderá a estar muy cerca de 0 o de 1, provocando que se sature. Este problema es similar al que ocurría en la capa de salida y que solucionamos eligiendo una mejor función costo. En este caso el problema es en la primer capa oculta y lo que haremos será elegir una mejor inicialización. Para eso basta con volver a examinar la ecuación (2.16). Haciendo $\operatorname{Var}[w_i] = 1/n$ vamos a asegurarnos de que $\operatorname{Var}[z] = 1$, logrando que la varianza de la salida lineal de la neurona se mantenga igual a la varianza de la entrada, sin importar su tamaño.

El análisis anterior aplica a la interfaz entre la capa de entrada y la primer capa oculta, pero cabe preguntarse qué ocurre con la varianza a medida que la información se propaga por el resto de las capas de la red. Antes asumimos que $x = a^0$ tenía media cero, pero eso ya no ocurre para $a^1, \ldots, a^n$ luego de haber aplicado la activación sigmoide, dado que tiene media 1/2. En el artículo [1] donde Glorot y Bengio introducen el método de inicialización que vamos a ver en esta sección, lo que ellos hacen es reemplazar la función sigmoide por otra similar: la tangente hiperbólica.

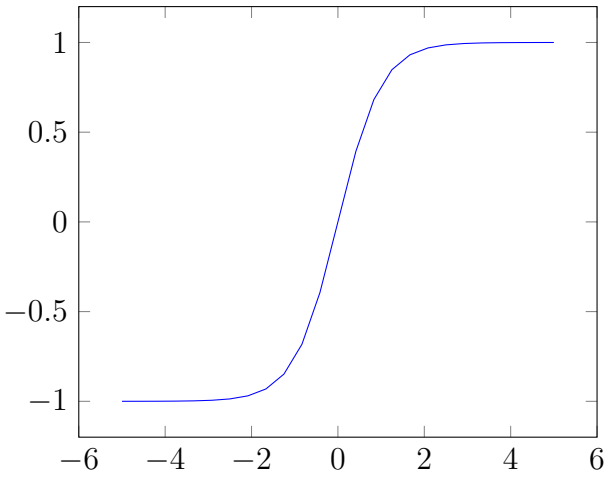


Figura 12: Tangente Hiperbólica

Esta función presenta un comportamiento similar al de la sigmoide, pero con la diferencia de que su media es cero, lo cual nos permite utilizar los cálculos hechos hasta ahora. Adaptando la ecuación (2.16) para el resto de capas, nos queda

$$\text{Var}\left[z^l\right] = \sum_k^{n_l} \text{Var}\left[w_k^l\right] \text{Var}\left[a_k^{l-1}\right] = n_l \,\text{Var}\left[w^l\right] \text{Var}\left[a^{l-1}\right]. \tag{2.18}$$

Si nos mantenemos dentro de la zona donde la neurona presenta un régimen lineal podemos asumir que $a^l = z^l$. La intención en la inicialización es elegir los pesos de tal manera que nos mantengamos en esa zona, ya que es ahí donde activaciones como la sigmoide o tanh presentan gradientes significativos, o en otras palabras, se encuentran lejos del punto de saturación. En ese caso, para una combinación de L capas, podemos deducir por recursión que

$$\text{Var}\left[z^L\right] = \text{Var}\left[x\right] \prod_{l=1}^{L} n_l \,\text{Var}\left[w_i^l\right]. \tag{2.19}$$

Nuestro objetivo al igual que antes es mantener la varianza de la salida igual a la varianza de la entrada, a fin de evitar aumentar o disminuir las amplitudes de las señales de manera exponencial. Una condición suficiente para lograr esto es que

$$\text{Var}\left[w^l\right] = 1/n_l \quad \text{para todo } l. \tag{2.20}$$

Por lo tanto, de acuerdo al criterio anterior, la estrategia de inicialización de los pesos sería utilizar una distribución normal de media cero y desvío estándar $1/\sqrt{n_l}$, referida a veces como *inicialización Lecun*.

Hasta ahora nos hemos preocupado del comportamiento de la varianza al propagar la información a través de la red hacia adelante, pero también nos puede ayudar mantener una señal alta en los gradientes durante la propagación hacia atrás. Para esto notemos que de acuerdo a las ecuación (1.36) de backprop tenemos que

$$\frac{\partial C}{\partial a^l} = (w^{l+1})^T \frac{\partial C}{\partial z^{l+1}}. \tag{2.21}$$

Si por simplicidad de notación hacemos $\alpha^l = \partial C/\partial a^l$ entonces la expresión anterior nos queda

$$\alpha^l = (w^{l+1})^T \delta^{l+1}. \tag{2.22}$$

También tenemos que

$$\delta^l = \alpha^l \sigma'(z^l). \tag{2.23}$$

De manera similar a lo hecho anteriormente, asumimos que δ^l y w^l son independientes entre sí para todo l, de modo que α^l tiene media cero cuando w^l es inicializado con una distribución simétrica alrededor del cero. Si ahora calculamos la varianza de α^l obtenemos

$$\text{Var}\left[\alpha^l\right] = n_{l+1} \text{Var}\left[w^{l+1}\right] \text{Var}\left[\delta^{l+1}\right]. \tag{2.24}$$

Dentro de la zona donde la neurona mantiene un régimen lineal tenemos que $\delta^l \approx \alpha^l$, de modo que

$$\text{Var}\left[\alpha^l\right] \approx n_{l+1} \text{Var}\left[w^{l+1}\right] \text{Var}\left[\alpha^{l+1}\right]. \tag{2.25}$$

Componiendo L capas resulta

$$\text{Var}\left[\alpha^1\right] \approx \text{Var}\left[\alpha^L\right] \prod_{l=1}^{L-1} n_{l+1} \text{Var}\left[w^{l+1}\right]. \tag{2.26}$$

Nuevamente, tomando $\text{Var}\left[w^l\right] = 1/n_{l+1}$ para todo l, logramos que $\text{Var}\left[\alpha^1\right] \approx \text{Var}\left[\alpha^L\right]$, con lo cual evitamos que los gradientes crezcan explosivamente o se desvanezcan.

En resumen, las condiciones que tenemos sobre los pesos hasta ahora son

$$\text{Var}\left[w^l\right] = 1/n_l, \tag{2.27}$$

$$\text{Var}\left[w^l\right] = 1/n_{l+1}. \tag{2.28}$$

Solo se pueden cumplir ambas al mismo tiempo si el número de neuronas de la capa l es igual al número de neuronas de la capa $l+1$. Esto no suele ocurrir, pero como compromiso se pueden promediar ambos valores, con lo cual el criterio de inicialización para los pesos consistiría en utilizar una distribución normal de media cero y desvío estándar $\sqrt{2/(n_l + n_{l+1})}$. Esto se suele conocer en la literatura como *inicialización Xavier*.

Ahora vamos a implementar estos dos métodos de inicialización para nuestra red actual. Para eso vamos a tener que introducir una nueva función de activación, que es la tanh. La biblioteca Numpy ya trae esta función predefinida, por lo que solo nos hace falta definir su derivada, como haremos a continuación:

```
def tanh_prime(x):
    return 1 - np.tanh(x)**2
```

Ahora necesitamos modificar las funciones de `feedforward` y `backprop` para que usen la nueva activación:

```
def feedforward(x):
    a[0] = x
    z[0] = x
    for k in range(1, L-1):
```

```
        z[k] = w[k] @ a[k-1] + b[k]
        a[k] = np.tanh(z[k])
    z[-1] = w[-1] @ a[-2] + b[-1]
    a[-1] = sigmoid(z[-1])
    return a[-1]

def backprop(x, y):
    feedforward(x)
    delta = (a[-1] - y)
    dw[-1] = delta @ a[-2].T
    db[-1] = np.sum(delta, axis=1, keepdims=True)
    for k in reversed(range(1, L-1)):
        delta = (w[k+1].T @ delta) * tanh_prime(z[k])
        dw[k] = delta @ a[k-1].T
        db[k] = np.sum(delta, axis=1, keepdims=True)
```

Notemos que hemos modificado las activaciones de todas las capas menos la última, donde seguimos usando la sigmoide. La razón de esto es que nos es conveniente que la salida siga manteniéndose entre 0 y 1, evitando tener que modificar los rótulos en el conjunto de entrenamiento. Esto también nos permite seguir usando la entropía cruzada como función costo tal como la definimos antes, sin cambios en el método de backprop. Otro motivo es que la función de salida esta relacionada a la elección del modelo probabilístico que se utiliza en la red, lo cual a su vez está relacionado con la elección de la función costo. En problemas de clasificación es habitual utilizar modelos Bernoulli o multinomiales, los cuales suelen ir emparejados con salidas sigmoides y softmax respectivamente. La función softmax es en esencia una salida sigmoide normalizada, de tal manera que describa una distribución de probabilidad. El costo que se usa con esta función es muy similar a la entropía cruzada usada en la sigmoide y mantiene las mismas propiedades para el backpropagation. Todo esto se verá más adelante, en la sección 2.4.

Ahora solo nos queda la implementación de los métodos de inicialización, que es como sigue:

```
def xavier_init():
    for k in range(1, L):
        in_size = neurons[k-1]
        out_size = neurons[k]
        w[k] = np.sqrt(2/(in_size+out_size)) * rnd.randn(out_size, in_size)
        b[k] = np.zeros((out_size, 1))

def lecun_init():
    for k in range(1, L):
        in_size = neurons[k-1]
        out_size = neurons[k]
        w[k] = np.sqrt(1/in_size) * rnd.randn(out_size, in_size)
        b[k] = np.zeros((out_size, 1))
```

Por último ejecutamos nuestro programa.

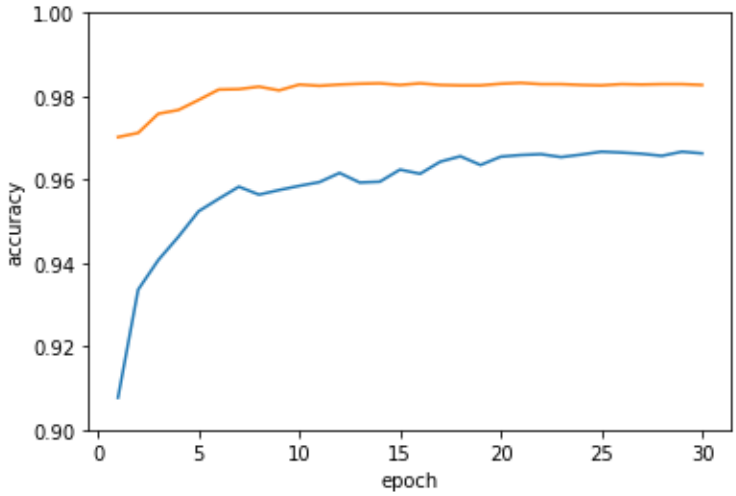


Figura 13: Comparación de la evolución de dos rondas de entrenamiento, una utilizando la inicialización normal estándar (azul) y la otra utilizando la inicialización Xavier (naranja).

```
trng_set, val_set = load_data()
xavier_init()
SGD(trng_set, val_set, cross_entropy_cost, epochs=30, bat_size=32, eta=0.1)
```

Los resultados durante el entrenamiento son los siguientes:

```
epoch:  1 | acc: 96.37% | loss: 0.24644
epoch:  2 | acc: 96.64% | loss: 0.22010
epoch:  3 | acc: 97.48% | loss: 0.17114
epoch:  4 | acc: 97.78% | loss: 0.15080
epoch:  5 | acc: 97.72% | loss: 0.14832
epoch:  6 | acc: 98.03% | loss: 0.14024
...
epoch: 26 | acc: 98.25% | loss: 0.14749
epoch: 27 | acc: 98.30% | loss: 0.14779
epoch: 28 | acc: 98.21% | loss: 0.14861
epoch: 29 | acc: 98.25% | loss: 0.14923
epoch: 30 | acc: 98.26% | loss: 0.14892
```

La mejora en el rendimiento de la red es evidente. La exactitud ya es de 96.37 % en la primera época, mientras que usando la inicialización anterior obteníamos 90.84 %. No solo eso, sino que la máxima exactitud obtenida durante el entrenamiento es superior a la que arribamos en ejecuciones anteriores: un 96.70 % contra un 98.30 % con la inicialización nueva. La mejora se puede explicar tanto por la inicialización como por la nueva elección en la función de activación. Sin embargo, si ahora hacemos una ejecución de nuestro programa revirtiendo todas las activaciones a la función sigmoide que veníamos usando previamente, los resultados durante el entrenamiento son los siguientes:

```
epoch:  1 | acc: 92.91% | loss: 0.47548
epoch:  2 | acc: 94.77% | loss: 0.35199
```

```
epoch:  3 | acc: 95.84% | loss: 0.27888
epoch:  4 | acc: 96.63% | loss: 0.24165
epoch:  5 | acc: 97.06% | loss: 0.20564
...
epoch: 26 | acc: 98.38% | loss: 0.12277
epoch: 27 | acc: 98.27% | loss: 0.12473
epoch: 28 | acc: 98.37% | loss: 0.12382
epoch: 29 | acc: 98.36% | loss: 0.12347
epoch: 30 | acc: 98.34% | loss: 0.12419
```

Podemos ver que con la sigmoide las primeras iteraciones no fueron tan buenas como con la tanh, pero la exactitud máxima obtenida es similar, presentando ambas un buen rendimiento, por lo que definitivamente la inicialización ayudó de manera notable. Recordemos que la teoría de inicialización que vimos antes está basada en una activación simétrica alrededor del cero, no siendo este el caso de la sigmoide. Esto explicaría que al principio la red aprenda mucho más rápido con la activación tanh y no tanto con la sigmoide. No obstante, independientemente del rendimiento en las primeras épocas, ambas se vieron beneficiadas durante el entrenamiento por esta nueva inicialización, lo cual es un poco curioso, teniendo en cuenta que la sigmoide no fue contemplada en la teoría.

Para explicar esto volvamos a la ecuación (2.13) y supongamos que tenemos una entrada fija x. En ese caso, suponiendo como hemos hecho hasta ahora que los pesos son i.i.d, tenemos que

$$\operatorname{Var}[z] = \operatorname{Var}[w] \sum_{k}^{n} x_k^2 \leq n \operatorname{Var}[w]. \tag{2.29}$$

De acuerdo a lo anterior, si elegimos $\operatorname{Var}[w] < 1/n$ logramos mantener bajo control la varianza de la salida lineal de la neurona, evitando que la distribución de los pesos sea muy ancha, lo cual daría lugar a que aparezcan valores demasiado altos, saturando las neuronas. Es posible que usando este método en el caso de la sigmoide, la varianza sea un poco más pequeña de lo ideal, pero en todo caso siempre será mejor mantener los pesos dentro de un rango más bien pequeño y no uno demasiado amplio, lo cual queda evidenciado empíricamente en los resultados anteriores.

Otro detalle que cabe mencionar es que en el caso de que la entrada para la red consista en imágenes, deja de ser cierto que los x_k, que en este caso serían los píxeles, sean i.i.d. Por lo tanto, una de las hipótesis usadas para hacer tratables los cálculos no es válida. Pese a esto, podemos ver la teoría expuesta anteriormente como una heurística que nos ayuda a inicializar los pesos de manera razonable, y que en la práctica se ha comprobado que ayuda a mejorar el aprendizaje de la red.

Nos queda probar la inicialización Lecun con activaciones tanh. Los resultados son los siguientes.

```
epoch:  1 | acc: 96.45% | loss: 0.23892
epoch:  2 | acc: 97.27% | loss: 0.17803
epoch:  3 | acc: 97.25% | loss: 0.17646
epoch:  4 | acc: 97.76% | loss: 0.14704
epoch:  5 | acc: 97.92% | loss: 0.14930
...
```

```
epoch: 26 | acc: 98.29% | loss: 0.14245
epoch: 27 | acc: 98.29% | loss: 0.14323
epoch: 28 | acc: 98.30% | loss: 0.14333
epoch: 29 | acc: 98.30% | loss: 0.14390
epoch: 30 | acc: 98.29% | loss: 0.14418
```

Como se puede ver, en nuestro caso no se observa una diferencia significativa entre la inicialización Xavier y Lecun, y se suele usar un criterio u otro de acuerdo a las preferencias y requerimientos particulares de la red.

2.3. Unidades Ocultas y Activaciones

En principio hemos construido nuestra red neuronal basada en unidades sigmoides, y de acuerdo al teorema de representación para redes neuronales, una red basada en este tipo de neuronas puede aproximar cualquier función continua sobre un compacto [2, 3]. Sin embargo, en la práctica, redes neuronales basadas en otro tipo de unidades pueden presentar un mejor rendimiento. Dependiendo de la aplicación, reemplazar la activación sigmoide puede permitir a la red aprender más rápido, generalizar mejor, o ambas.

Una de las alternativas para la unidad sigmoide ya la vimos, y es la neurona tanh, la cual nos permitió, combinada con una inicialización apropiada, mejorar notablemente la velocidad de aprendizaje de la red. Esta neurona está estrechamente relacionada con la sigmoide. Para ver esto, basta con calcular

$$\begin{aligned}
\frac{1+\tanh(z/2)}{2} &= \frac{1}{2}\left[1+\frac{e^{z/2}-e^{-z/2}}{e^{z/2}+e^{-z/2}}\right] \\
&= \frac{1}{2}\frac{e^{z/2}+e^{-z/2}+e^{z/2}-e^{-z/2}}{e^{z/2}+e^{-z/2}} \\
&= \frac{e^{z/2}}{e^{z/2}+e^{-z/2}} \\
&= \frac{1}{1+e^{-z}} \\
&= \sigma(z).
\end{aligned} \tag{2.30}$$

Por lo tanto, la activación tanh no es más que una versión reescalada de la activación sigmoide. También se puede apreciar la similitud comparando las figuras 3b y 12. Una ventaja de la tangente hiperbólica frente la sigmoide es que se parece más a la identidad cerca del 0, en el sentido de que $\tanh(0) = 0$ y $\tanh'(0) = 1$, mientras que $\sigma(0) = 1/2$ y $\sigma'(0) = 1/4$. Esto hace que entrenar una red neuronal utilizando unidades tanh sea similar a entrenar un modelo lineal, siempre que las activaciones se mantengan pequeñas, lo cual hace el entrenamiento más fácil.

Pese a que para nuestro problema ambos tipos de activaciones han tenido un rendimiento satisfactorio, estas sufren de un problema importante en el contexto más general de redes neuronales, y es la tendencia a saturarse en la mayor parte del dominio, tanto si la magnitud de su entrada z es muy alta como también si es muy baja, siendo más sensibles solo cuando z se encuentra cerca de 0. Esta tendencia hace que en muchos otros problemas el aprendizaje basado en gradiente se vuelva difícil y es por esta razón que hoy en día se suelen usar otras activaciones para las unidades ocultas. Por otra parte,

su uso para las unidades de salida es compatible con el aprendizaje basado en gradiente cuando se combinan con una función costo que elimina el fenómeno de saturación sobre la capa de salida, que es precisamente lo que hicimos al introducir la entropía cruzada como función costo en la sección 2.1.

Para combatir el problema presente en las unidades sigmoides y tanh es común utilizar hoy en día las *unidades rectificadas lineales (ReLU)* y sus generalizaciones. La activación utilizada en las ReLU se define como

$$g(z) = \text{máx}\left\{0, z\right\}. \tag{2.31}$$

Estás unidades son fáciles de optimizar porque son similares a las unidades lineales, haciendo que los gradientes se mantengan grandes siempre que la unidad esté activa.

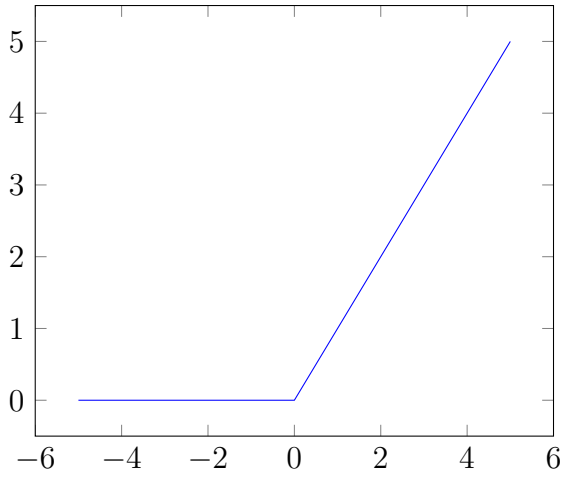


Figura 14: ReLU

Dado que la activación suele aplicarse sobre una transformación afín (como hemos hecho hasta ahora), resulta conveniente inicializar los sesgos a un valor positivo pequeño como 0.1, para así asegurarse de que las ReLU estén activas para la mayor parte de las entradas del conjunto de entrenamiento y obtener señal en los gradientes.

Al utilizar las ReLU se elimina el problema de saturación que se podía observar en las unidades sigmoides, pero por otra parte estas no pueden aprender cuando su entrada es negativa, dado que en ese caso el gradiente se desvanece. De hecho, algo que puede ocurrir durante el entrenamiento al utilizar este tipo de unidades es que caigan en estados donde quedan desactivadas para cualquier entrada, y al ser cero el gradiente estas no pueden recuperarse, quedando efectivamente "muertas". Para remediar este inconveniente existen varias generalizaciones de estas unidades. Una alternativa es usar una pendiente distinta de cero cuando $z_i < 0$, quedando la activación de la forma

$$a_i = g(z, \alpha)_i = \text{máx}(0, z_i) + \alpha_i \,\text{mín}(0, z_i). \tag{2.32}$$

Si fijamos α_i a un valor pequeño como 0.01 obtenemos las que se conocen como *leaky ReLU*. Por otra parte, se puede tratar α_i como un parámetro que se aprende, de la misma manera que los pesos y los sesgos. En ese caso estamos en presencia de las *ReLU paramétricas*.

Otra posibilidad es la utilización de las unidades *maxout*. Estas llevan la generalización de las ReLU un paso más allá, ya que lo que hacen es calcular, dada una entrada x,

$$h_i(x) = \max_{j \in [1,k]} z_{ij}, \tag{2.33}$$

donde

$$z_{ij} = W_{ij:}x + b_{ij}, \tag{2.34}$$

y $W \in \mathbb{R}^{d \times m \times k}$ y $b \in \mathbb{R}^{m \times k}$ son parámetros que se aprenden. Notemos, a modo de comparación, que hasta ahora lo que hacíamos era calcular $z_i = W_{i:}x + b_i$ y luego aplicarle la activación, mientras que ahora estamos calculando k funciones lineales distintas para una entrada x, quedándonos luego con el máximo en cada una de las componentes. Lo que estamos diciendo con esto es que la unidad maxout puede describir una función lineal a trozos, con hasta k piezas. Esto permite que durante el entrenamiento la red pueda aprender cual es la función de activación que debería usar, siendo posible aproximar cualquier función convexa con tal de tomar k lo suficientemente grande. En particular, se puede ver que una capa maxout con dos piezas puede aprender a implementar una activación ReLU, Leaky ReLU o ReLU paramétrica, pero también puede aprender funciones completamente distintas. Por otra parte, el problema que surge con las unidades maxout es que presentan k veces la cantidad de parámetros que si se usara una activación convencional, haciéndolas computacionalmente más costosas y siendo necesaria más regularización para prevenir el overfitting (el concepto de regularización se trata más adelante, en la sección 3).

2.4. Unidades de Salida, Modelos y Costos

Lo dicho hasta ahora se ha enfocado más que nada en aquellas unidades que son utilizadas en las capas ocultas, pero hay algunas unidades que suelen ser destinadas para la salida de la red neuronal.

En general, la red neuronal se puede ver como un modelo paramétrico que define una distribución $p(y \mid x, \theta)$. En el caso de nuestro clasificador, esta cuantifica la probabilidad de que habiendo observado una entrada x, la clase que le corresponde sea y, condicionado a los parámetros θ. Supongamos ahora que tenemos un conjunto de entrenamiento $T = (X, Y)$, donde $X = \{x_1, \ldots, x_n\}$ es el conjunto de todas nuestras entradas, $Y = \{y_1, \ldots, y_n\}$ es el conjunto que contiene los respectivos rótulos, y donde suponemos que cada par (x_k, y_k) fue muestreado de manera independiente desde una distribución p que desconocemos. Lo que nos interesa durante el entrenamiento es encontrar aquellos parámetros θ que hacen que la red asigne una mayor probabilidad a aquél rótulo que realmente le corresponde a la entrada. En otros términos, lo que queremos es encontrar

$$\theta_{ML} = \arg\max_{\theta} p(Y \mid X, \theta); \tag{2.35}$$

esto se conoce como *estimación por máxima verosimilitud*, o MLE por sus siglas en inglés.

Como estamos asumiendo que los ejemplos del conjunto de entrenamiento son i.i.d, la formula anterior se puede descomponer como

$$\theta_{ML} = \arg\max_{\theta} \prod_{i=1}^{n} p(y_i \mid x_i, \theta). \tag{2.36}$$

Este producto sobre muchas probabilidades presenta ciertos inconvenientes. Uno de ellos es que al estar multiplicando por cantidades entre 0 y 1, va a tener tendencia al *underflow* numérico. Esto quiere decir que la magnitud del producto se hace tan pequeña que supera la precisión de la máquina para representarlo, con lo cual pasa a ser 0. Para remediar esto, lo que se puede hacer es maximizar el logaritmo del producto, lo cual no cambia la solución de nuestro problema de optimización, y de esta manera conseguimos reemplazar el producto por una suma:

$$\theta_{ML} = \arg\max_{\theta} \sum_{i=1}^{n} \log p(y_i \mid x_i, \theta). \tag{2.37}$$

También podemos dividir por n sin modificar el arg max, obteniendo así una expresión en términos de la esperanza respecto a la distribución empírica definida por el conjunto de entrenamiento. En ambos casos lo que estamos haciendo es maximizar el *log-likelihood*. Por último, como conveniencia al trabajar con bibliotecas numéricas, se suele expresar el problema en términos de minimización, quedando

$$\theta_{ML} = \arg\min_{\theta} \sum_{i=1}^{n} -\log p(y_i \mid x_i, \theta), \tag{2.38}$$

donde ahora minimizamos el *negative log-likelihood*, habitualmente abreviado como NLL.

La forma de la función costo cambia de acuerdo al modelo que se este usando, es decir, en función de la forma que tenga $\log p$. Por ejemplo, si hacemos $p(y \mid x) = \mathcal{N}(y, f(x, \theta), I)$ recuperamos el costo cuadrático. Para ver eso, denotemos con μ_i a la salida $f(x_i, \theta)$ y calculemos:

$$\begin{aligned}
\sum_{i=1}^{n} -\log p(y_i \mid x_i, \theta) &= \sum_{i=1}^{n} -\log \left(\frac{\exp\left(-\frac{1}{2}(y_i - \mu_i)^t I^{-1} (y_i - \mu_i)\right)}{\sqrt{(2\pi)^k \, |I|}} \right) \\
&= \sum_{i=1}^{n} -\log \left(\exp\left(-\frac{1}{2} \left\| y_i - \mu_i \right\|_2^2 \right) \right) \\
&+ \sum_{i=1}^{n} -\log \left(-\sqrt{(2\pi)^k \, |I|} \right) \\
&= \sum_{i=1}^{n} \frac{1}{2} \left\| y_i - f(x_i, \theta) \right\|_2^2 + C,
\end{aligned} \tag{2.39}$$

donde la constante puede ser descartada, ya que no depende de θ. Lo que esto nos dice es que minimizar el error cuadrático es lo mismo que realizar estimación por máxima verosimilitud para un modelo Gaussiano.

Un modelo que se suele usar habitualmente en tareas de clasificación que requieren predecir el valor de una variable binaria es el modelo Bernoulli, cuya distribución de probabilidad viene dada por

$$\mathrm{Ber}(y \mid \mu) = \mu^y (1 - \mu)^{1-y} \quad \text{para } y \in \{0, 1\}, \tag{2.40}$$

donde μ es un parámetro que representa la probabilidad de éxito y que también se puede interpretar como la media. Ese parámetro va a estar dado por $f(x, \theta)$, ya que lo que nos

interesa es calcular la probabilidad de que el rótulo sea 0 o 1 de acuerdo al valor que nos devuelva la red. Nuestro modelo es entonces

$$P(y \mid x, \theta) = \text{Ber}(y \mid f(x, \theta)) \tag{2.41}$$

y también tenemos que

$$P(y = 1 \mid x, \theta) = f(x, \theta). \tag{2.42}$$

Para que $f(x, \theta)$ sea una probabilidad vamos a tener que asegurarnos de que la salida de la red se encuentre en el intervalo $[0, 1]$. Es evidente que utilizando la salida lineal $w \cdot x$ eso no se cumple, pero si utilizamos la activación sigmoide para reducir la salida al intervalo $(0, 1)$ podemos utilizar el modelo sin problemas. Esto motiva la elección de la unidad sigmoide como salida de la red para problemas de clasificación, independientemente de la activación que se esté utilizando para las unidades ocultas.

Si ahora calculamos el NLL para este modelo, lo que tenemos es

$$\begin{aligned}
\sum_{i=1}^{n} -\log p(y_i \mid x_i, \theta) &= -\sum_{i=1}^{n} \log \left[\text{Ber}(y \mid f(x, \theta))\right] \\
&= -\sum_{i=1}^{n} \log \left[f(x_i, \theta)^{y_i}(1 - f(x, \theta))^{1-y_i}\right] \\
&= -\sum_{i=1}^{n} y_i \log f(x_i, \theta) + (1 - y_i) \log(1 - f(x_i, \theta)),
\end{aligned} \tag{2.43}$$

obteniendo así la entropía cruzada, que vimos antes en la sección 2.1. Podemos ver entonces que minimizar la entropía cruzada es equivalente a realizar estimación por máxima verosimilitud con un modelo Bernoulli.

Es posible combinar varios clasificadores binarios para obtener un clasificador que pueda trabajar con varias clases. De hecho, eso es lo que hemos hecho hasta ahora con nuestra red neuronal. Lo que tenemos son varios modelos Bernoulli con salida sigmoide, caracterizados por

$$p_k(y = 1 \mid x, \theta) = \frac{\exp(z_k(x, \theta))}{1 + \exp(z_k(x, \theta))} = \sigma(z_k(x, \theta)) = f_k(x, \theta), \tag{2.44}$$

donde $z(x, \theta)$ es la salida lineal de la última capa de la red. En el caso de MNIST, dada una entrada x, $f_1(x)$ nos va a devolver la probabilidad de que x sea un 1, mientras que $f_2(x)$ nos va a devolver la probabilidad de que x sea un 2, y así sucesivamente. Sin embargo, el vector en su conjunto no representa una distribución de probabilidad, sino que en cierta forma cada elemento individual lo es.

Para problemas de clasificación con varias clases existe otra posibilidad, que es usar un modelo multinomial, dado por

$$\text{Mul}(y \mid \mu, n) = \binom{n}{y_1 \dots y_K} \prod_{k=1}^{K} \mu_k^{y_k}. \tag{2.45}$$

donde n es el número de experimentos aleatorios y K el número de clases. En nuestro caso tiene sentido hacer $n = 1$, de la misma manera que antes elegimos un modelo

Bernoulli (binomial con $n = 1$). Para usar este modelo es necesario adaptar la ecuación (2.44) del modelo Bernoulli con salida sigmoide, normalizando el vector de salida para que represente una distribución de probabilidad, obteniendo la fórmula

$$p(y = k \mid x, \theta) = \frac{\exp(z_k(x, \theta))}{\sum_j \exp(z_j(x, \theta))} = \text{softmax}(z(x, \theta))_k = f_k(x, \theta). \tag{2.46}$$

Si ahora aplicamos el NLL a este modelo, para un solo ejemplo de entrenamiento x, con $\mu_k = f_k(x, \theta)$, obtenemos la función costo

$$\begin{aligned} C(x, \theta) = -\log\left(\prod_{k=1}^{K} \mu_k^{y_k}\right) = -\sum_{k=1}^{K} \log(\mu_k^{y_k}) = -\sum_{k=1}^{K} y_k \log(\mu_k) \\ = -\sum_{k=1}^{k} y_k \log(\text{softmax}(z)_k). \end{aligned} \tag{2.47}$$

Por lo que el costo promedio sobre todo el conjunto de entrenamiento de tamaño m es

$$C(\theta) = -\frac{1}{m} \sum_{k,j} y_{kj} \log(\text{softmax}(z_{kj})), \tag{2.48}$$

siendo este el equivalente de la entropía cruzada, pero para salida softmax y modelo multinomial.

En cuanto al algoritmo de backpropagation, las unidades softmax con costo NLL se comportan exactamente igual que las sigmoides con la entropía cruzada, siendo el gradiente de la capa de salida igual a $f(x, \theta) - y$.

2.5. Hiperparámetros y conjuntos de validación

La mayoría de los algoritmos para redes neuronales utilizan hiperparámetros. Estos son parámetros que controlan el comportamiento del modelo pero que no son aprendidos por el algoritmo, al contrario de lo que ocurre con los pesos y sesgos de nuestra red. En ocasiones, un determinado parámetro se determina que sea un hiperparámetro debido a que este es difícil de optimizar, pero con más frecuencia lo que ocurre es que no es apropiado que el parámetro en cuestión sea configurado a partir del conjunto de entrenamiento[2].

Dado que los hiperparámetros deben ser configurados a partir de un conjunto que el algoritmo de entrenamiento no observe, la opción obvia es configurar los hiperparámetros basándonos en el rendimiento obtenido sobre el conjunto de prueba. No obstante, existe un problema con este enfoque, y es que uno puede terminar encontrando hiperparámetros que se ajustan a peculiaridades del conjunto de prueba, pero que luego no generalizan a otros conjuntos de datos. Lo anterior no es más que otro caso de overfitting, pero esta vez causado por los hiperparámetros en vez de los parámetros. La solución a esto es utilizar un *conjunto de validación*. Este conjunto no es más que un subconjunto de los datos

[2]Esto en particular aplica a todos los hiperparámetros que controlan la capacidad del modelo, ya que en ese caso el algoritmo de entrenamiento tenderá a elegir la máxima capacidad posible, lo cual lograría el objetivo de reducir el costo de entrenamiento, pero daría lugar al overfitting. Ver sección 3.

de entrenamiento que uno aparta para ser usado exclusivamente con el fin de validar los hiperparámetros. De acuerdo a este esquema se utiliza el conjunto de entrenamiento para elegir los parámetros, el conjunto de validación para elegir los hiperparámetros, y el conjunto de prueba para realizar una evaluación final del rendimiento de la red neuronal una vez que todos los hiperparámetros fueron configurados. Usualmente uno suele usar un 80 % de los datos de entrenamiento con fines de entrenamiento y el 20 % restante con fines de validación.

Por otra parte, dividir el conjunto de datos en un conjunto de entrenamiento y de prueba fijos puede ser problemático si resulta en un conjunto de prueba pequeño, ya que nos generaría incertidumbre estadística respecto al error de prueba estimado, haciendo difícil afirmar, por ejemplo, que un algoritmo A funciona mejor que un algoritmo B en una tarea dada. Cuando el conjunto de datos es pequeño, existen procedimientos alternativos que nos permiten usar todos los ejemplos en la estimación del error medio de prueba, a cambio de un mayor costo computacional. La idea es repetir el entrenamiento y la prueba en subconjuntos elegidos aleatoriamente o en particiones de los datos originales. Uno de los métodos más comunes es la validación cruzada, en la cual se particiona el conjunto de datos en k subconjuntos disjuntos. El error de prueba se puede estimar entonces tomando el error promedio sobre k experimentos, donde en el experimento j se utiliza el j-ésimo conjunto como conjunto de prueba y el resto como conjunto de entrenamiento.

Algo que no se ha mencionado hasta ahora es qué maneras existen de elegir los hiperparámetros. La manera más básica consiste en realizar una búsqueda manual basándonos en el monitoreo del proceso de entrenamiento y en los resultados obtenidos sobre el conjunto de validación. Una ventaja de esto es que la búsqueda manual puede dar cierta intuición sobre la respuesta de la red frente a los cambios en los hiperparámetros. Sin embargo, este procedimiento por prueba y error puede consumir demasiado tiempo.

Otra posibilidad es utilizar un algoritmo para realizar una búsqueda exhaustiva sobre un subconjunto del espacio de hiperparámetros, en lo que se conoce como *grid search*. Esta búsqueda debe estar guiada por algún tipo de métrica de rendimiento, como puede ser la validación cruzada o la evaluación sobre un conjunto de validación previamente apartado. Un ejemplo de la aplicación de este método es la elección de la tasa de aprendizaje η y el parámetro de regularización λ para el algoritmo de SGD. Ambos parámetros son continuos, por lo que uno define conjuntos discretos con valores razonables para ambos, como puede ser

$$\eta \in \{1; 0.1; 0.01; 0.001\},$$
$$\lambda \in \{0.1; 0.2; 0.5; 1.0\}.$$

Lo que hace grid search es entrenar la red utilizando como hiperparámetros cada punto del producto cartesiano de los conjuntos anteriores, evaluando luego el rendimiento y quedándose con aquella tupla de hiperparámetros que logró el mejor desempeño. Este enfoque suele funcionar cuando el número de dimensiones es pequeño, preferentemente menor o igual que 4. Para más dimensiones se sufre de la maldición de la dimensionalidad, requiriendo una cantidad excesivamente grande de puntos para cubrir adecuadamente el espacio de búsqueda. La consecuencia de esto es que el tiempo de búsqueda aumenta de manera explosiva (usualmente de manera exponencial) con el aumento de las dimensiones. Por otra parte, este método es simple de implementar y fácilmente paralelizable, y si uno

dispone de suficiente poder computacional suele encontrar mejores parámetros que los que una búsqueda manual podría encontrar en la misma cantidad de tiempo.

Otra posibilidad es realizar una búsqueda aleatoria o *random search*. En este esquema, en vez de realizar una búsqueda exhaustiva sobre un subconjunto del espacio de hiperparámetros, lo que se hace en cambio es elegirlos de manera aleatoria. Aunque en principio pueda parecer anti-intuitivo, este procedimiento es superior a grid search, particularmente cuando el número de dimensiones es alto. Una exposición sobre esta cuestión se puede ver en [4]. En la figura 15 se puede apreciar la intuición detrás del mejor desempeño al utilizar random search. Mientras que en grid search se repiten los hiperparámetros que uno prueba en cada dimensión, en random search es extremadamente improbable que estos se repitan, lo cual permite en el mismo tiempo de búsqueda tener más diversidad en las configuraciones que uno examina. Esto a su vez posibilita una exploración más amplia del espacio de hiperparámetros.

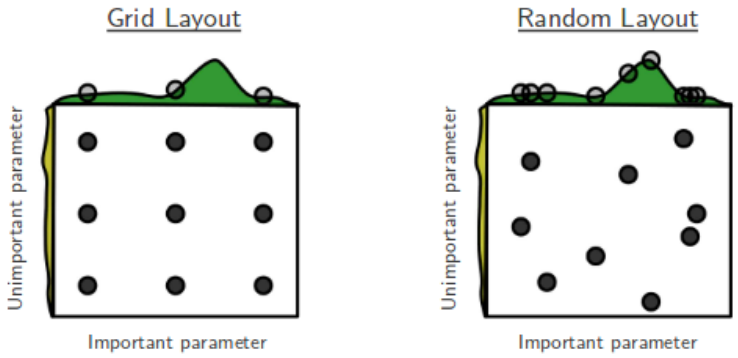


Figura 15: Comparación entre grid search y random search.

2.6. Reimplementación del MLP

En esta sección vamos a modificar y expandir un poco la implementación para poder trabajar mejor con lo visto hasta ahora. El primer cambio que vamos a hacer es en la definición de la red. Previamente la habíamos definido en base a ciertas variables globales que gobernaban su arquitectura y almacenaban la información. Todo esto puede ser encapsulado dentro de una clase, permitiéndonos trabajar la red neuronal como si fuera un objeto. El código es como sigue:

```
#%% Arquitectura de la red

class Net:
    def __init__(self, neurons):
        self.neurons = neurons
        self.L = len(neurons)
        self.w, self.b, self.dw, self.db, self.a, self.z = \
        [self.L*[0] for k in range(6)]

    def feedforward(self, x):
```

```
        self.a[0] = x
        self.z[0] = x
        for k in range(1, self.L-1):
            self.z[k] = self.w[k] @ self.a[k-1] + self.b[k]
            self.a[k] = Relu.f(self.z[k])
        self.z[-1] = self.w[-1] @ self.a[-2] + self.b[-1]
        self.a[-1] = Softmax.f(self.z[-1])
        return self.a[-1]

    def backprop(self, x, y):
        self.feedforward(x)
        delta = (self.a[-1] - y)
        self.dw[-1] = delta @ self.a[-2].T
        self.db[-1] = np.sum(delta, axis=1, keepdims=True)
        for k in reversed(range(1, self.L-1)):
            delta = (self.w[k+1].T @ delta) * Relu.g(self.z[k])
            self.dw[k] = delta @ self.a[k-1].T
            self.db[k] = np.sum(delta, axis=1, keepdims=True)
```

Primero tenemos un método de inicialización que hace el trabajo de definir las variables que guardan los parámetros, las activaciones y los gradientes. Esto es prácticamente igual al código utilizado en la primera implementación para definir esas mismas variables, solo que adaptado a la sintaxis de clases. Luego tenemos un método que define la función de feedforward de la red, que es igual a la rutina del mismo nombre en la implementación anterior. Por último tenemos un método que define la manera en que se propaga la información hacia atrás y nos devuelve los gradientes; este método también es igual al de la implementación anterior, salvo por algunas diferencias de sintaxis. Queda de esta manera definida la red como un objeto que consta de pesos y activaciones, asociados a dos métodos que definen su arquitectura y funcionamiento. Modificando la rutina de feedforward y backpropagation se define la arquitectura de la red neuronal.

La mayoría de las funciones que utilizamos en la implementación original solo requieren leves variaciones en la sintaxis para acomodar la nueva definición de la red neuronal, permaneciendo el resto exactamente igual, por lo que a continuación solo nos vamos a ocupar de las nuevas definiciones.

Originalmente trabajábamos las activaciones utilizando dos funciones por separado: la activación propiamente dicha y su gradiente. Algo que se puede hacer es combinar estas dos en una clase, para así mantenerlas unificadas. Las distintas activaciones quedan definidas como sigue:

```
#%% Activaciones

class Sigmoid:
    @staticmethod
    def f(x):
        s = 1/(1 + np.exp(-x))
        return s

    @staticmethod
    def g(x):
        s = 1/(1 + np.exp(-x))
        return s * (1 - s)

class Tanh:
    @staticmethod
    def f(x):
        return np.tanh(x)

    @staticmethod
    def g(x):
        return 1 - np.tanh(x)**2

class Relu:
    @staticmethod
    def f(x):
        xx = x.copy()
        xx[xx<0] = 0
        return xx

    @staticmethod
    def g(x):
        xx = x.copy()
        xx[xx<=0] = 0
        xx[xx>0] = 1
        return xx

class Softmax:
    @staticmethod
    def f(x):
        s = np.exp(x - np.max(x, axis=0))
        return s/np.sum(s, axis=0)

    @staticmethod
    def g(x):
        s = np.exp(x - np.max(x, axis=0))
        s = s / np.sum(s, axis=0)
        I = np.eye(x.shape[0])
        return s.T * (I - s)
```

Cada clase cuenta con dos métodos: uno es f, que representa la activación, y el otro es g, que nos devuelve el gradiente. En esta lista ahora también encontramos la definición para las unidades ReLU utilizadas en las capas ocultas y la activación softmax utilizada en la capa de salida, según lo visto en las secciones 2.3 y 2.4. Algo a notar es que dado que la activación softmax solo nos interesa para la capa de salida y emparejada junto con el costo NLL, solo usaremos la función f, no siendo necesario el gradiente g. En nuestra implementación del método de backpropagation este gradiente ya fue cancelado al calcular δ para la primer capa, al igual que ocurre al utilizar la sigmoide junto a la entropía cruzada. Pese a esto, incorporamos el gradiente g en la definición de la Softmax por completitud, con la salvedad de que no puede ser usado en procesamiento por lotes, sino que sirve para ejemplos individuales.

Para las unidades ReLU existe una variación del esquema de inicialización que vimos en la sección 2.2, denominada inicialización *He*.

```
def he_init(net):
    for k in range(1, net.L):
        in_size = net.neurons[k-1]
        out_size = net.neurons[k]
        net.w[k] = np.sqrt(2/in_size) * rnd.randn(out_size, in_size)
        net.b[k] = 0.1*np.ones((out_size, 1))
```

Esta es, en esencia, una adaptación de la inicialización Xavier pero para redes basadas en ReLU. Un desarrollo de este método de inicialización se puede ver en [5].

Para el entrenamiento necesitamos alguna función costo que optimizar. Las que hemos visto hasta ahora son las siguientes

```
#%% Costos

def quad_cost(a, y):
    return 1/2 * np.sum((a - y)**2)

def cross_entropy_cost(a, y):
    return -np.sum(np.nan_to_num(y*np.log(a) + (1 - y)*np.log(1 - a)))

def nll_cost(a, y):
    return -np.sum(y * np.log(a))
```

La nueva adición de esta lista es el costo NLL, definido para ser usado con un modelo Multinoulli, de acuerdo a lo visto en la ecuación (2.48) de la sección 2.4.

La arquitectura de la red está configurada en este momento para utilizar la nuevas unidades ReLU y la salida softmax. Procederemos entonces a entrenar la red y ver los resultados. Para eso cargamos los datos, construimos la red, inicializamos los parámetros y ejecutamos el optimizador. La red será entrenada con el costo NLL como objetivo, por 30 épocas, con un minilote de tamaño 32 y tasa de aprendizaje 0.1.

```
trng_set, val_set = load_data()
neurons = [784, 400, 100, 10]
net = Net(neurons)
```

```
he_init(net)
SGD(net, trng_set, val_set, nll_cost,
    epochs=30, bat_size=32, rate=0.1)
```

Los resultados son los siguientes:

```
epoch:  1 | acc: 96.77% | loss: 0.10601
epoch:  2 | acc: 97.15% | loss: 0.09750
epoch:  3 | acc: 97.50% | loss: 0.08935
epoch:  4 | acc: 97.45% | loss: 0.08835
epoch:  5 | acc: 97.69% | loss: 0.08753
...
epoch: 26 | acc: 98.20% | loss: 0.09810
epoch: 27 | acc: 98.22% | loss: 0.09849
epoch: 28 | acc: 98.19% | loss: 0.09909
epoch: 29 | acc: 98.23% | loss: 0.09926
epoch: 30 | acc: 98.20% | loss: 0.09950
```

La combinación ReLU-softmax nos ha dado un rendimiento similar al que ya habíamos obtenido antes para nuestro problema. Si ahora reemplazamos la capa softmax con una capa sigmoide y utilizamos la entropía cruzada como costo, los resultados son:

```
epoch:  1 | acc: 94.78% | loss: 0.37706
epoch:  2 | acc: 96.05% | loss: 0.27801
epoch:  3 | acc: 96.79% | loss: 0.24017
epoch:  4 | acc: 97.06% | loss: 0.21110
epoch:  5 | acc: 97.29% | loss: 0.19685
...
epoch: 26 | acc: 98.22% | loss: 0.16491
epoch: 27 | acc: 98.18% | loss: 0.16732
epoch: 28 | acc: 98.26% | loss: 0.16714
epoch: 29 | acc: 98.20% | loss: 0.16778
epoch: 30 | acc: 98.24% | loss: 0.16926
```

Seguimos sin mejorar la exactitud y pareciera que ya estamos llegando a una barrera cerca del 98.3 %, lo cual nos hace sospechar que podemos estar cerca del límite de exactitud que nos ofrece la arquitectura actual para la red neuronal. En las secciones que siguen exploraremos las posibilidades que existen para superar esta barrera. En particular, la diferencia principal se logrará al utilizar las denominadas redes convolucionales, que se verán en la sección 5.

3. Regularización

3.1. Overfitting, Underfitting y Capacidad

El desafío central al trabajar con redes neuronales es que nuestro algoritmo debe tener buen rendimiento sobre entradas nuevas no observadas previamente, y no solo en aquellas para las cuales nuestro modelo fue entrenado. Esta habilidad de poder desempeñarse correctamente sobre entradas nuevas se denomina *generalización*.

Típicamente, en un modelo de redes neuronales, tenemos acceso a un conjunto de entrenamiento y podemos calcular alguna medida del error sobre este conjunto, denominada error de entrenamiento. Lo que separa a machine learning y deep learning de un mero problema de optimización es que esperamos que el error de generalización sea pequeño también. Este error se define como la esperanza del error sobre una nueva entrada. Esta esperanza se toma sobre diferentes entradas posibles, extraídas a partir de la distribución de entradas que esperamos que el sistema encuentre en la práctica. Es usual estimar el error de generalización del modelo midiendo su rendimiento sobre un conjunto de prueba conteniendo ejemplos obtenidos separadamente del conjunto de entrenamiento.

Los factores que determinan cuan bien va a desempeñarse un modelo de redes neuronales son su capacidad de hacer pequeño el error de entrenamiento, y de hacer la brecha entre el error de entrenamiento y el error de prueba lo más pequeña posible. El primer factor se corresponde con el concepto de *underfitting* (subajuste), que es cuando el modelo no puede obtener un error suficientemente bajo sobre el conjunto de entrenamiento, mientras que el segundo factor se corresponde con el de *overfitting* (sobreajuste), que ocurre cuando la brecha entre el error de entrenamiento y el error de prueba es demasiado grande.

Es posible controlar la tendencia de un modelo a presentar overfitting o underfitting alterando su *capacidad*. La capacidad de un modelo es, informalmente hablando, su habilidad de ajustarse a un número amplio de funciones. Un modelo con una baja capacidad puede tener problemas para ajustarse al conjunto de entrenamiento, mientras que un modelo con capacidad alta puede ser que aprenda detalles del conjunto de entrenamiento que luego no le sirven para el conjunto de prueba. Los algoritmos de aprendizaje generalmente van a presentar mejor rendimiento cuando la capacidad del modelo concuerda con la complejidad real del problema y la cantidad de datos de entrenamiento que se poseen. Modelos con capacidad insuficiente no pueden resolver tareas complejas, mientras que si la capacidad es demasiado alta, pueden presentar overfitting.

Una manera de controlar la capacidad es eligiendo el espacio de hipótesis, que consiste del conjunto de funciones que el algoritmo puede elegir como solución. Como ejemplo, en una regresión lineal uno tiene al espacio de funciones lineales como espacio de hipótesis. Si uno extiende esta familia de funciones para incluir a los polinomios, uno está aumentando el espacio de hipótesis y por ende la capacidad del modelo. A esto se lo conoce como *capacidad representacional* del modelo. Notemos que el comportamiento del modelo no solo depende del tamaño del espacio de hipótesis, sino también del tipo de funciones que contiene. Es evidente que las funciones lineales pueden ser útiles para aquellos problemas en los cuales la relación entre la entrada y la salida es cercana a lineal, pero no tanto para problemas con comportamiento no lineal. Nuevamente, la elección del espacio de hipótesis es importante en estos casos.

Muchas veces encontrar la mejor función dentro de esta familia de funciones es un problema de optimización difícil, por lo que en la práctica el algoritmo de aprendizaje no necesariamente encuentra la mejor función, sino una que reduzca significativamente el error de entrenamiento. Estas limitaciones, como las imperfecciones en el algoritmo de aprendizaje, implican que la *capacidad efectiva* del algoritmo puede ser menor que la capacidad representacional.

Otra forma en la que se puede controlar la capacidad del modelo es mediante la preferencia de una solución sobre otra en el espacio de hipótesis. Por ejemplo, en el caso de una regresión lineal, uno puede preferir soluciones con parámetros pequeños, o en el caso de nuestra red, uno puede querer que la magnitud de los pesos se mantenga dentro de ciertos límites. Esto se puede ver como una manera más general de controlar la capacidad que incluir o excluir miembros del espacio de hipótesis, ya que uno puede pensar la exclusión de una función del espacio de hipótesis como expresar una preferencia infinitamente fuerte contra esa función.

Existen muchas maneras de expresar preferencias por distintas soluciones, ya sea de manera explicita o implícita, y en conjunto estos enfoques se conocen como regularización. Otra forma en la que se puede ver la regularización es como cualquier modificación que uno realice sobre un algoritmo de aprendizaje de tal manera que se reduzca su error de generalización pero no su error de entrenamiento, que es lo mismo que decir que uno muestra preferencia por aquellas soluciones que disminuyen el error de generalización, siendo esto lo que en el fondo nos interesa durante el aprendizaje.

3.2. Regularización L^2

Muchos enfoques de regularización se basan en limitar la capacidad de los modelos por medio de un término Ω que penaliza la norma de los parámetros, el cual se suma a la función objetivo C, de tal manera que

$$\bar{C}(x, y, \theta) = C(x, y, \theta) + \lambda\Omega(\theta), \tag{3.1}$$

donde λ es un hiperparámetro que regula la contribución de Ω relativa a la función objetivo C. Si uno hace $\lambda = 0$ se anula la regularización, mientras que valores más altos resultan en mayor regularización. La idea detrás de esto es que al momento de entrenar la red se tendrá que optimizar la función objetivo $\bar{C}$, para lo cual será necesario que decrezcan tanto C como $\Omega(\theta)$. Diferentes elecciones de Ω y del parámetro λ pueden hacer que soluciones que eran óptimas para la función costo original C ahora sean penalizadas de tal manera que otras soluciones pasen a ser preferidas.

Una posibilidad para el término de regularización Ω es utilizar la norma L^2 sobre los pesos, por lo que nuestra función costo quedaría

$$\bar{C}(x, y, \theta) = C(x, y, \theta) + \frac{\lambda}{2} \|w\|_2^2 . \tag{3.2}$$

Esto hace que la red prefiera aprender pesos pequeños, solo permitiendo que sean grandes si mejoraran notablemente el costo original. En otras palabras, es un compromiso entre aprender pesos pequeños y minimizar el costo.

Si los pesos de la red son pequeños, eso significa que el comportamiento de la red no va a cambiar mucho al modificar aleatoriamente algunas entradas, lo cual hace difícil

para la red aprender los efectos del ruido local en los datos. Dicho de otra manera, uno está haciendo que piezas de información individuales no tengan demasiada influencia en la salida de la red. En contraposición, si los pesos de la red son grandes, esta puede modificar su comportamiento drásticamente en respuesta a pequeños cambios en la entrada. La consecuencia de esto es que una red no regularizada puede usar pesos grandes para aprender modelos complejos que acarrean mucha información sobre el ruido en los datos de entrenamiento.

En redes neuronales es usual elegir una penalización para la norma de los parámetros que solo penaliza los pesos w utilizados en la transformación afín de cada capa, dejando los sesgos b sin regularizar. La razón de esto es que tener valores muy grandes para b no hace que la neurona sea sensible a sus entradas en la misma manera que tener pesos w muy grandes. Además, regularizar los parámetros del sesgo puede introducir una cantidad significativa de underfitting.

Podemos ganar cierta idea del comportamiento de la regularización L^2 si estudiamos el gradiente de la función objetivo regularizada. Si calculamos las derivadas parciales en la ecuación (3.2) obtenemos

$$\nabla_w \bar{C} = \nabla_w C + \lambda w, \tag{3.3}$$
$$\nabla_b \bar{C} = \nabla_b C. \tag{3.4}$$

En consecuencia, utilizando descenso por gradiente, la regla para actualizar los sesgos no cambia, pero para los pesos pasa a ser

$$\begin{aligned} w_{k+1} &= w_k - \eta \nabla_w \bar{C}(w_k) \\ &= w_k - \eta \nabla_w C(w_k) - \eta\lambda w_k \\ &= (1 - \eta\lambda) w_k - \eta \nabla_w C(w_k). \end{aligned} \tag{3.5}$$

Esto no es muy diferente que la regla usual usada en descenso por gradiente, excepto que escalamos los pesos w por el factor $(1-\eta\lambda)$ antes de actualizar, haciéndolos más pequeños. Es por esta razón que también se suele llamar *weight decay* (que se podría traducir como "decaimiento de pesos") a este tipo de regularización, aunque cabe destacar que esto no necesariamente es válido para otros algoritmos que no sean descenso por gradiente, para los cuales la regularización L^2 y weight decay pueden ser distintas.

Hemos descripto lo que ocurre durante un solo paso de actualización, pero cabe preguntarse qué ocurre durante todo el entrenamiento. Para esto vamos a simplificar el análisis haciendo una aproximación cuadrática de la función objetivo en el entorno del valor de los pesos donde se obtiene el mínimo costo no regularizado, al que llamaremos $w^* = \arg\min_w C(w)$. La aproximación $\hat{C}$ está dada por

$$\hat{C}(\theta) = C(w^*) + \frac{1}{2}(w - w^*)^t H (w - w^*), \tag{3.6}$$

donde H es el hessiano de la matriz C con respecto a w, evaluado en w^*. No hay término de primer orden en esta aproximación cuadrática porque w^* está definido como el mínimo, donde el gradiente se desvanece. De manera similar, dado que w^* es la ubicación de un mínimo de C, podemos concluir que H es semidefinida positiva.

Para estudiar el efecto de la regularización L^2, vamos a modificar la ecuación

$$\nabla_w \hat{C}(w) = H(w - w^*), \tag{3.7}$$

agregándole el término extra λw. Ahora podemos resolver para el mínimo de la versión regularizada de $\hat{C}$, donde usaremos la variable $\bar{w}$ para representar la ubicación del mínimo:

$$\begin{aligned}\lambda\bar{w} + H(\bar{w} - w^*) &= 0 \\ (H + \lambda I)\bar{w} &= Hw^* \\ \bar{w} &= (H + \lambda I)^{-1} Hw^*.\end{aligned}$$

A medida que λ se acerca a cero, la solución regularizada $\bar{w}$ se acerca a w^*. Por otra parte, para ver qué ocurre cuando λ crece, notemos primero que H es real y simétrica, por lo que podemos descomponerla en una matriz diagonal D y una base ortonormal de autovectores Q, tal que $H = QDQ^t$. Aplicando la descomposición a la ecuación anterior, obtenemos

$$\begin{aligned}\bar{w} &= (QDQ^t + \lambda I)^{-1} QDQ^t w^* \\ &= \left[Q(D + \lambda I)Q^t\right]^{-1} QDQ^t w^* \\ &= Q(D + \lambda I)^{-1} DQ^t w^*.\end{aligned} \tag{3.8}$$

Vemos que el efecto de la regularización es reescalar w^* a lo largo de los ejes definidos por los autovectores de H. Específicamente, la componente de w^* que está alineada con el i-ésimo autovalor de H es reescalada por un factor de $\frac{\alpha_i}{\alpha_i + \lambda}$, donde α_i es el i-ésimo autovalor de H.

A lo largo de las direcciones donde los autovalores de H son relativamente grandes, por ejemplo donde $\alpha_i \gg \lambda$, el efecto de la regularización es relativamente pequeño. Por otra parte, las componentes con $\alpha_i \ll \lambda$ serán encogidas hasta tener una magnitud cercana a cero. Solo las direcciones a lo largo de las cuales los parámetros contribuyen de manera significativa a reducir la función objetivo son preservadas relativamente intactas. En direcciones que no contribuyen a reducir la función objetivo, los valores pequeños del Hessiano nos indican que el movimiento a lo largo de esas trayectorias no va a incrementar significativamente el gradiente. Componentes del vector de pesos correspondiendo a direcciones de poca importancia como esas son decaídos por medio del uso de la regularización durante el entrenamiento.

Para implementar la regularización L^2 en nuestro código basta con modificar la línea que actualiza los parámetros en el algoritmo de SGD para incorporar la ecuación (3.5).

```
decay = (1 - rate*weight_decay/trng_size)
net.w[k] =  decay * net.w[k] - rate/bat_size * net.dw[k]
net.b[k] =  net.b[k] - rate/bat_size * net.db[k]
```

El algoritmo de SGD recibe ahora un parámetro `weight_decay`, con el cual regulamos el grado de regularización que queremos. Este se usa para calcular el valor de la variable `decay`, que contiene el factor de decaimiento, por el que luego multiplicamos a w en el paso de actualización, de acuerdo a la ecuación (3.5).

3.3. Regularización L^1

La regularización L^2 suele ser la forma más común de penalización sobre el tamaño de los parámetros, pero hay otras opciones. Una de ellas es usar regularización L^1. En

este tipo de regularización, el término de penalización está dado por

$$\Omega(\theta) = \|w\|_1 = \sum_i |w_i|\,, \tag{3.9}$$

por lo que la función objetivo regularizada es

$$\bar{C}(x, y, w) = C(x, y, w) + \lambda \|w\|_1\,, \tag{3.10}$$

con su correspondiente gradiente

$$\nabla_w \bar{C} = \lambda \operatorname{sgn} w + \nabla_w C. \tag{3.11}$$

De acuerdo a esto, la regla para actualizar los pesos utilizando descenso por gradiente es

$$w_{k+1} = w_k - \eta\lambda \operatorname{sgn} w_k - \eta \nabla_w C(w_k). \tag{3.12}$$

Podemos comparar esto con la expresión para regularización L^2, que era

$$w_{k+1} = (1 - \eta\lambda) w_k - \eta \nabla_w C(w_k). \tag{3.13}$$

En ambas expresiones el efecto de la regularización es encoger los pesos, pero la manera en que lo hacen es diferente. En la regularización L^1 los pesos son encogidos por una cantidad constante, mientras que en la regularización L^2 los pesos son disminuidos por una cantidad que es proporcional a w. Por lo tanto, cuando un peso en particular tiene una magnitud muy grande, la regularización L^1 lo disminuye menos que la regularización L^2. Por otra parte, si la magnitud del peso es pequeña, L^1 lo disminuye mucho más que L^2. El resultado es que la regularización L^1 tiende a favorecer una solución más dispersa, ya que tiende a concentrar el peso de la red en un número pequeño de conexiones importantes, mientras que el resto de los pesos tienden a cero.

3.4. Aumento de Datos

La mejor manera de hacer que una red neuronal generalice mejor es entrenarla en un número mayor de datos. En la práctica la cantidad de datos que tenemos es limitada, por lo que una manera de esquivar este problema es aumentar nuestros datos artificialmente. Supongamos que tenemos un clasificador, el cual debe tomar una entrada x, que podría ser una imagen o video, y resumirla en una sola categoría y. Esto significa que el clasificador debe ser invariante a un número amplio de transformaciones sobre la entrada x, por lo que uno puede generar nuevos pares (x, y) aplicando estas transformaciones sobre las entradas x del conjunto de entrenamiento, las cuales uno luego agrega al conjunto. Para ver un ejemplo, supongamos que x es una imagen. Si la trasladamos, la giramos o invertimos, en general la clase correcta con la que uno identifica la imagen no va a cambiar, por lo que uno puede aplicar estas operaciones para aumentar los datos de entrenamiento. Uno debe tener cuidado, sin embargo, de no aplicar transformaciones que pudieran cambiar la clase correcta, lo cual puede ocurrir si uno aplica rotaciones de 180 grados en MNIST, por ejemplo, haciendo que un 6 se transforme en 9 o viceversa.

3.5. Early Stopping

Al entrenar modelos grandes con suficiente capacidad representacional como para presentar overfitting, suele ocurrir que el error de entrenamiento decrece de manera constante con el tiempo, pero en algún punto el error de validación empieza a aumentar otra vez. En este caso, sería mejor si volvemos en el tiempo a aquellos parámetros con el mejor error de validación. Lo que se puede hacer es guardar una copia de los parámetros de la red cada vez que el error de validación mejora. El algoritmo de entrenamiento termina cuando el error no mejora por un número prefijado de iteraciones, y cuando esto ocurre regresamos la copia de los parámetros que tenemos almacenada, con los cuales hemos obtenido el mejor error de validación hasta el momento. Esta estrategia se conoce como *early stopping* y es uno de los métodos de regularización comúnmente usados con redes neuronales, tanto por su simplicidad como por su eficacia.

Si pensamos el número de pasos de entrenamiento como un hiperparámetro, early stopping no es más que una forma de seleccionar este hiperparámetro. También es una forma de controlar la capacidad efectiva del modelo, determinando cuantos pasos puede tomar el algoritmo para ajustar el conjunto de entrenamiento. Mientras que la mayoría de los hiperparámetros se eligen por medio de un proceso de prueba y error, seleccionándolos al comienzo del entrenamiento y viendo su efecto a través de varios pasos, la configuración del tiempo de entrenamiento es peculiar en que una sola ronda de entrenamiento prueba varios valores de este hiperparámetro. Un costo de usar early stopping para elegir su valor es tener que evaluar periódicamente la red sobre el conjunto de validación, cosa que uno hace de todas formas si se está monitoreando el entrenamiento. Otro costo es el de tener que mantener una copia de los mejores parámetros, el cual suele ser despreciable dada la capacidad de almacenamiento disponible en el hardware actual.

Una ventaja de esta estrategia como forma de regularización es que no requiere cambios en el procedimiento de entrenamiento, la función objetivo, o sobre la red, por lo que es fácil de utilizar sin modificar la dinámica del aprendizaje. También puede ser usada en conjunto con otras formas de regularización.

Dado un valor inicial de los parámetros θ_0, early stopping tiene el efecto de restringir el procedimiento de optimización a un entorno de θ_0 relativamente pequeño en el espacio de parámetros. Para ser un poco más precisos, si tomamos t pasos de optimización con una tasa de aprendizaje η, entonces podemos ver el producto ηt como una medida de la capacidad efectiva del modelo. Si asumimos que el gradiente es acotado, restringiendo el número de iteraciones y la tasa de aprendizaje lo que estamos logrando es limitar el volumen de espacio que se puede alcanzar desde θ_0. Es en este sentido que ηt se comporta de manera similar al recíproco del coeficiente usado en la regularización L^2, justificando el considerar early stopping como una técnica de regularización. De hecho, se puede mostrar que en el caso de un modelo lineal con función de error cuadrática y utilizando descenso por gradiente, early stopping es equivalente a regularización L^2. Para eso ver [6].

3.6. Bagging y Métodos de Ensamble

La idea de estas técnicas es entrenar diferentes modelos por separado y después hacer que todos los modelos voten una salida para cada ejemplo de prueba. Este es un ejemplo de una estrategia general denominada *promediado de modelos*, y las técnicas que emplean esta estrategia se suelen denominar *métodos de ensamble*. Para ver por qué esto permite

reducir el error de generalización, consideremos un conjunto de k modelos de regresión. Supongamos que cada modelo comete un error ϵ_i en cada ejemplo, donde el error sigue una distribución normal multivariada con varianzas $\mathrm{E}\left[\epsilon_i^2\right] = v$ y covarianzas $\mathrm{E}\left[\epsilon_i \epsilon_j\right] = c$. En ese caso el error incurrido por la predicción promedio de todos los modelos es $\frac{1}{k}\sum_i \epsilon_i$, mientras que el error cuadrático esperado es

$$\mathrm{E}\left[\frac{1}{k}\left(\sum_i \epsilon_i\right)^2\right] = \frac{1}{k^2}\mathrm{E}\left[\sum_i\left(\epsilon_i^2 + \sum_{j\neq i}\epsilon_i\epsilon_j\right)\right] = \frac{1}{k}v + \frac{k-1}{k}c. \tag{3.14}$$

En el caso en que los errores tengan correlación perfecta, el error cuadrático medio se reduce a v, por lo que no logramos ninguna mejora. Por otra parte, en el caso en que los errores no presenten correlación alguna, la esperanza sobre el error cuadrático pasa a ser $\frac{v}{k}$, por lo que el error disminuye de manera lineal con el número k de modelos que se utilizan. La conclusión es que el ensamble de modelos va a funcionar al menos tan bien como cualquiera de sus miembros, y en el caso de que los miembros realicen errores independientes, el rendimiento va a ser superior al de sus miembros.

Existen distintas formas de construir el ensamble de modelos. Una posibilidad puede ser que cada miembro del ensamble consista de un modelo completamente distinto utilizando un algoritmo o función objetivo diferente. En el caso del método de bagging, lo que se hace es lo opuesto, reutilizando varias veces el mismo modelo, algoritmo de entrenamiento y función objetivo. Específicamente, se construyen k conjuntos de datos distintos, cada uno con el mismo número de ejemplos que el conjunto original, pero muestreados con reemplazo. Luego se entrena el modelo i sobre el conjunto i. Las diferencias entre los ejemplos incluidos en cada conjunto de datos resultan en la diferencia entre los modelos una vez entrenados.

Dado que las variaciones debido a la inicialización aleatoria y la selección al azar de los minilotes o los hiperparámetros pueden hacer que una misma red neuronal alcance una variedad amplia de soluciones, estas suelen beneficiarse del promediado de modelos incluso si todos son entrenados sobre el mismo conjunto.

3.7. Dropout

Esta técnica consiste en entrenar el ensamble de modelos conformado por todas las subredes que pueden formarse removiendo unidades de una red base (menos las de salida). Dado que la mayoría de las redes neuronales que se trabajan suelen estar basadas en una serie de transformaciones afines y activaciones no lineales, generalmente es posible remover una unidad de la red multiplicando su salida por cero. Para entrenar con dropout se usa un algoritmo basado en minilotes, como gradiente estocástico. Cada vez que cargamos un ejemplo al minilote se elige de manera aleatoria una máscara binaria, la cual aplicamos a todas las entradas y unidades ocultas de la red. Luego se realizan los pasos de feedforward, backpropagation y actualización de los parámetros como siempre. La máscara para cada unidad es escogida de manera independiente y la probabilidad de elegir una máscara con un valor de uno es un hiperparámetro fijado antes del entrenamiento.

Formalmente, supongamos que un vector máscara μ especifica qué unidades incluir y $C(\theta, \mu)$ define el costo del modelo definido por los parámetros θ y máscara μ. El entrenamiento por dropout consiste en minimizar $\mathrm{E}_\mu\left[C(\theta, \mu)\right]$. La esperanza contiene una

cantidad exponencial de términos, pero se puede obtener un estimador no sesgado de su gradiente muestreando valores de μ.

Dropout se puede puede pensar como una especie de bagging con una cantidad exponencial de modelos, pero no son exactamente lo mismo. En el caso de bagging los modelos son independientes, mientras que en el caso de dropout los modelos comparten parámetros, con cada modelo heredando un subconjunto diferente de parámetros de la red base. El hecho de que se compartan los parámetros hace que sea posible representar una cantidad exponencial de modelos con una cantidad tratable de memoria. Otra diferencia es que en el caso de bagging, cada modelo es entrenado en su respectivo conjunto de entrenamiento, mientras que para dropout los modelos no suelen ser entrenados explícitamente. Usualmente el modelo es lo suficientemente grande como para hacer imposible extraer todas las posibles subredes dentro de un tiempo razonable. En cambio, una fracción de las posibles subredes se entrenan en cada época individual del algoritmo, y el hecho de compartir parámetros hace que las subredes restantes arriben a buenas configuraciones para los parámetros. Más allá de estas diferencias, el algoritmo de dropout se comporta como un algoritmo de bagging.

Ya vimos que para realizar una predicción, un ensamble de redes acumula votos de todos sus miembros, a lo cual nos referiremos como *inferencia* en este contexto. Supongamos, como suele ocurrir al trabajar con redes, que la salida es una distribución de probabilidad. En el caso de bagging, cada modelo produce una distribución de probabilidad $p_i(y \mid x)$, por lo que la predicción del ensamble está dada por la media aritmética de estas distribuciones

$$\frac{1}{k}\sum_{i=1}^{k} p_i(y \mid x). \tag{3.15}$$

En el caso de dropout, cada subred está definida por una mascara μ, definiendo una distribución $p(y \mid x, \mu)$. La media aritmética sobre todas las máscaras está dada por

$$\sum_{\mu} p(\mu)p(y \mid x, \mu), \tag{3.16}$$

donde $p(\mu)$ es la distribución que fue usada para muestrear μ en tiempo de entrenamiento. Dado que esta suma incluye un número exponencial de términos, no es posible evaluarla excepto cuando la estructura del modelo permite alguna forma de simplificación. Lo que se puede hacer en cambio es aproximar la inferencia promediando la salida de varias máscaras.

Otro enfoque consiste en usar la media geométrica en vez de la media aritmética. Para garantizar que esta sea una distribución de probabilidad, imponemos el requerimiento de que ninguno de los submodelos asigne probabilidad 0 a algún evento y renormalizamos la distribución resultante. La distribución sin normalizar queda definida entonces como

$$\hat{p}(y \mid x) = \sqrt[2^d]{\prod_{\mu} p(y \mid x, \mu)}, \tag{3.17}$$

donde d es el número de unidades que pueden ser descartadas, mientras que la distribución normalizada es

$$p(y \mid x) = \frac{\hat{p}(y \mid x)}{\sum_{y'} \hat{p}(y' \mid x)}. \tag{3.18}$$

La idea principal detrás de este enfoque es que es posible aproximar la distribución de probabilidad evaluando $p(y \mid x)$ en un solo modelo, que consiste de la red completa con todas sus unidades, pero con los pesos de cada unidad multiplicados por la probabilidad de que dicha unidad sea incluida. La intuición detrás de esto es que así uno se asegura de que la esperanza sobre el valor total de la entrada a una unidad durante el tiempo de prueba sea más o menos similar a la esperanza de la entrada total a esa unidad durante el entrenamiento, independientemente de que se este entrenando con menos unidades. Debido a que generalmente se usa una probabilidad de inclusión de 1/2, la regla anterior suele reducirse a dividir los pesos por 2 al final del entrenamiento y después usar el modelo de la manera habitual.

Para poner un ejemplo de la regla de escalados de pesos, consideremos un clasificador basado en regresión softmax con n variables:

$$p(Y = y \mid x) = \text{softmax}(Wx + b)_y. \tag{3.19}$$

Es posible indexar la familia de submodelos multiplicando elemento a elemento por medio de un vector binario μ:

$$p(Y = y \mid x, \mu) = \text{softmax}(W(\mu \odot x) + b)_y. \tag{3.20}$$

Para ver que en este caso la regla de escalado de pesos es exacta, podemos examinar $\hat{p}$:

$$\begin{aligned} \hat{p}(Y = y \mid x) &= \sqrt[2^n]{\prod_\mu p(Y = y \mid x, \mu)} \\ &= \sqrt[2^n]{\prod_\mu \text{softmax}(W(\mu \odot x) + b)_y} \\ &= \sqrt[2^n]{\prod_\mu \frac{\exp(W_y(\mu \odot x) + b_y)}{\sum_{y'} \exp(W'_y(\mu \odot x) + b_{y'})}}. \end{aligned} \tag{3.21}$$

Dado que $\hat{p}$ será normalizado, podemos ignorar los factores que son constantes respecto a y:

$$\begin{aligned} \hat{p}(Y = y \mid x) &\propto \sqrt[2^n]{\prod_\mu \exp\left(W_y(\mu \odot x) + b_y\right)} \\ &= \exp\left(\frac{1}{2^n} \sum_d W_y(\mu \odot x) + b_y\right) \\ &= \exp\left(\frac{1}{2} W_y x + b_y\right). \end{aligned} \tag{3.22}$$

Obtenemos entonces un clasificador softmax con pesos $\frac{1}{2}W$.

La regla de escalado de pesos también es exacta en otros contextos, como redes con capas ocultas lineales, pero es solo una aproximación cuando hay activaciones no lineales presentes. En estos casos la aproximación no se encuentra caracterizada teóricamente de manera completa, pero empíricamente se sabe que funciona bien.

Una ventaja de dropout es que es computacionalmente económico, ya que para generar n números binarios aleatorios y multiplicarlos por el estado solo se requieren $O(n)$ cálculos por cada ejemplo por actualización, mientras que realizar inferencias desde el modelo ya entrenado tiene el mismo costo por ejemplo que si dropout no fuera usado. Otra ventaja es que no limita significativamente el tipo de modelo o procedimiento de entrenamiento que puede ser usado.

Hasta ahora hemos descripto dropout como una forma eficiente de realizar un bagging aproximado utilizando una cantidad exponencial de subredes. Otra manera es verlo como un ensamble de modelos que comparten unidades ocultas. Esto significa que cada unidad oculta debe poder desempeñarse correctamente independientemente de cuales sean las otras unidades en el modelo. Originalmente el desarrollo de dropout fue inspirado por una idea de la biología. La reproducción sexual, que involucra el intercambio de genes entre dos organismos diferentes, fomenta desde un punto de vista evolutivo que los genes no sean solo buenos, sino que también puedan trabajar bien con un conjunto aleatorio de otros genes, haciéndolos más robustos. Dado que el gen no puede aprender a confiar en la presencia fija de un número grande de compañeros, tiene que aprender a realizar algo útil por su cuenta o en colaboración con un número pequeño de otros genes. De manera similar ocurre con dropout, haciendo que las unidades ocultas sean más robustas y no dependan de otras para corregir sus errores. Esto previene que las unidades aprendan características inusuales del conjunto de entrenamiento, ya que deben ser capaces de aprender no solo características buenas, sino aquellas que son buenas en una variedad de contextos.

4. Optimización

Al trabajar con redes neuronales lo que realmente nos interesa optimizar durante el entrenamiento es alguna medida P que representa el rendimiento de la red. El inconveniente que surge es que P suele ser intratable y depende del conjunto de prueba, no del conjunto de entrenamiento. En consecuencia, lo que se hace es optimizar P de manera indirecta, recurriendo a otra función costo L, con la esperanza de que esto también mejore P.

El objetivo de un algoritmo de aprendizaje es reducir el error de generalización, que podemos escribir como

$$R(\theta) = \mathrm{E}_p \left[L(f(x, \theta), y) \right], \tag{4.1}$$

donde p es la distribución de los pares (x, y), L es la función que cuantifica la pérdida para cada par y $f(x, \theta)$ es la salida de la red. Esta cantidad no es más que la pérdida en la que se incurre para un determinado valor de los parámetros θ y es por esto que también se suele llamar *riesgo*. Generalmente no conocemos p, sino que tenemos un conjunto de entrenamiento, por lo que minimizamos el costo sobre este conjunto en vez de sobre toda la distribución, con lo cual ahora estamos minimizando el riesgo empírico

$$J(\theta) = \mathrm{E}_{\hat{p}} \left[L(f(x, \theta), y) \right] = \frac{1}{n} \sum_{k=1}^{n} L(f(x_k, \theta), y_k), \tag{4.2}$$

y esperamos que esto a su vez nos permita disminuir $R(\theta)$ de manera significativa. Sin embargo, este enfoque es propenso al overfitting, ya que modelos con alta capacidad pueden simplemente memorizar el conjunto de entrenamiento. Además, muchas técnicas efectivas de optimización se basan en descenso por gradiente, mientras que funciones como la que usamos para cuantificar la exactitud de la red no tienen derivadas útiles. Debido a esto es raro que se recurra a la minimización directa del riesgo empírico en redes neuronales, sino que se utilizan funciones substituto para la pérdida, como pueden ser el costo cuadrático, la entropía cruzada, o el negative log-likelihood, que son las que hemos visto hasta ahora.

4.1. Descenso por Gradiente Estocástico

A la hora de minimizar la función objetivo, una cantidad que nos va a interesar es el gradiente del costo:

$$\nabla_\theta C(\theta) = \mathrm{E}_{\hat{p}} \left[\nabla_\theta L(f(x, \theta), y) \right]. \tag{4.3}$$

Un detalle que surge es que evaluar esta esperanza de manera exacta es muy costoso si el conjunto de entrenamiento es muy grande. Sin embargo, es posible estimar la esperanza por medio de una muestra de n elementos tomados del conjunto de entrenamiento, con error de estimación dado por $\sigma/\sqrt{n}$, donde σ es el desvío estándar poblacional de las muestras. De acuerdo a la formula anterior, si suponemos que tenemos dos estimadores para el gradiente, uno basado en 100 ejemplos y otro basado en 10000, resulta que el último requiere 100 veces más cálculos que el primero, pero reduce el error estándar por solamente un factor de 10. Es por esto que es conveniente estimar el gradiente con una muestra en vez de calcularlo de manera exacta.

Los algoritmos de optimización que usan todo el conjunto de entrenamiento se suelen denominar *determinísticos*, mientras que aquellos que utilizan un subconjunto pequeño con algunos ejemplos (minilote) se denominan métodos estocásticos. El ejemplo canónico de este tipo de algoritmos es el método de descenso por gradiente estocástico (SGD).

El algoritmo en esencia sería el siguiente:

$\theta = \theta_0$;
mientras *no se cumpla criterio de parada* **hacer**
  extraer una muestra de m ejemplos del conjunto de entrenamiento;
  calcular estimador del gradiente: $g = \frac{1}{m} \sum_k^m \nabla_\theta L(f(x_k, \theta), y_k)$;
  actualizar los parámetros: $\theta = \theta - \eta g$
fin

Se suele extraer las muestras sin reposición y de manera secuencial, por lo que es habitual mezclar el conjunto de entrenamiento y volver a iterar una vez que ya se usaron todos los ejemplos, en lo que constituye una *época*. Si consideramos el número de épocas como un hiperparámetro, entonces el algoritmo anterior se puede escribir como:

$\theta = \theta_0$;
para *n entre 1 y máximo número de épocas* **hacer**
  mezclar el conjunto de entrenamiento;
  para *cada minilote de tamaño m del conjunto de entrenamiento* **hacer**
    calcular estimador del gradiente: $g = \frac{1}{m} \sum_k^m \nabla_\theta L(f(x_k, \theta), y_k)$;
    actualizar los parámetros: $\theta = \theta - \eta g$;
  fin
fin

El código de gradiente estocástico que hemos implementado (con regularización L^2) sigue este último patrón:

```
def SGD(net, trng_set, val_set, loss_fn,
        epochs=30, bat_size=100, rate=1, weight_decay=0):
    x, y = trng_set
    trng_size = x.shape[1]
    for n in range(epochs):
        shf_idx = rnd.permutation(trng_size)
        x, y = x[:, shf_idx], y[:, shf_idx]
        for k in range(0, trng_size, bat_size):
            net.backprop(x[:, k:k+bat_size], y[:, k:k+bat_size])
            for k in range(1, net.L):
                decay = (1 - rate*weight_decay/trng_size)
                net.w[k] =  decay * net.w[k] - rate/bat_size * net.dw[k]
                net.b[k] =  net.b[k] - rate/bat_size * net.db[k]
        acc, loss = performance(net, val_set, loss_fn)
        print('epoch: %2d | acc: %2.2f%% | loss: %.5f' % (n+1, acc*100, loss))
```

Es necesario otro bucle para el paso de actualización porque debemos actualizar los parámetros por cada capa, pero salvo por eso el código es esencialmente el mismo.

Un parámetro crucial para SGD es la tasa de aprendizaje. Si uno usara el gradiente completo de la función costo, como en el caso de descenso por gradiente, este decrecería en norma hasta finalmente hacerse cero a medida que nos acercamos al mínimo. Con SGD, por el contrario, uno introduce una fuente de ruido al estimar el gradiente, la cual no se desvanece cuando arribamos al mínimo. Hasta ahora hemos usado una tasa de aprendizaje fija, pero debido a lo anterior suele ser necesario decrecerla gradualmente, por lo que pasamos a tener una tasa de aprendizaje η_k, donde k denota el número de iteración. Unas condiciónes suficientes para garantizar la convergencia de SGD, conocidas como condiciones de *Robbins-Monro* [7], son las siguientes:

$$\begin{aligned} \sum_{k=1}^{\infty} \eta_k &= \infty, \\ \sum_{k=1}^{\infty} \eta_k^2 &< \infty. \end{aligned} \tag{4.4}$$

Algo que se suele hacer es decaer la tasa de manera lineal hasta la iteración t por medio de

$$\eta_k = (1 - \alpha_k)\eta_0 + \alpha_k \eta_t, \tag{4.5}$$

con $\alpha_k = \frac{k}{t}$. Luego de la iteración t se deja constante.

Una propiedad importante del algoritmo de SGD y derivados es que el tiempo de cálculo por actualización no aumenta con el número de ejemplos de entrenamiento. El número de actualizaciones requeridas para alcanzar la convergencia generalmente aumenta con el tamaño del conjunto de entrenamiento, pero a medida que dicho tamaño se acerca a infinito, el modelo eventualmente converge a su mejor error posible antes de pasar por cada ejemplo del conjunto. Desde este punto de vista se puede interpretar que el costo asintótico de entrenar un modelo con SGD es $O(1)$.

4.2. SGD + Momentum

El método de SGD puede ser lento a veces, pero existen maneras de acelerarlo. Si se piensa el espacio de parámetros en términos físicos, el método de descenso por gradiente consiste, en esencia, en desplazarse cuesta abajo sobre la superficie determinada por la función costo, hasta llegar al lugar más hondo. En el método de momentum esta analogía se hace más fuerte. Supongamos que nuestro vector de parámetros θ es una particula en un espacio de parámetros. Si esta particula experimenta una fuerza f esto causará que acelere, de acuerdo a la ecuación

$$f(t) = \frac{\partial^2 \theta(t)}{\partial t^2}, \tag{4.6}$$

que se puede reescribir como un sistema de ecuaciones de primer orden si introducimos una variable v representando la velocidad:

$$v(t) = \frac{\partial \theta(t)}{\partial t}, \tag{4.7}$$

$$f(t) = \frac{\partial v(t)}{\partial t}. \tag{4.8}$$

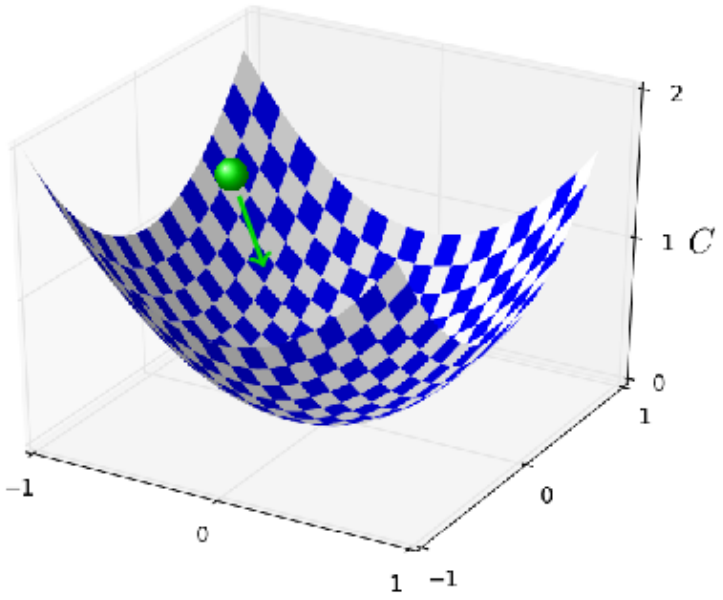


Figura 16: Se puede pensar el algoritmo de descenso por gradiente con momentum como una bola rodando sobre la superficie determinada por el costo, sujeta a fuerzas como la gravedad y la fricción, las cuales la llevan hacia el mínimo.

Una de las fuerzas actuando sobre θ sería el gradiente negativo del costo, $-\nabla_\theta C(\theta)$, que mueve a la partícula cuesta abajo por la superficie. Pero hace falta otra fuerza, ya que sino la partícula podría adquirir demasiado momentum, haciendo imposible que llegue al reposo cerca del mínimo. Esta otra fuerza es la fricción, proporcional a $-v(t)$, que en la analogía correspondería a la fricción viscosa. Existen otras posibilidades, como usar una fuerza proporcional a $-v^2(t)$, como en el caso de la fricción turbulenta, o una fuerza constante, como en el caso de la fricción seca. Sin embargo, la primera opción nos da una fuerza muy débil cuando la velocidad no es lo suficientemente alta, con lo cual la partícula difícilmente se detenga en el mínimo, mientras que la segunda opción es demasiado fuerte, lo cual frenaría la partícula antes de poder alcanzar el mínimo.

De acuerdo a lo anterior, el algoritmo de momentum no sería más que la solución numérica del sistema anterior, que se puede realizar por el método de Euler, tomando pequeños pasos finitos en la dirección de cada gradiente. El resultado son las siguientes ecuaciones para actualizar los parámetros θ:

$$v_k \leftarrow \alpha v - \eta \nabla_\theta \left(\frac{1}{m} \sum_k^m L(f(x_k, \theta), y) \right), \tag{4.9}$$

$$\theta \leftarrow \theta + v. \tag{4.10}$$

De la primera ecuación se puede ver que la velocidad se va incrementando de acuerdo al gradiente, pero al mismo tiempo se disminuye de acuerdo al parámetro α, que regula la

cantidad de fricción. Luego en la segunda ecuación se actualizan los parámetros según la velocidad. Los valores habituales para α son 0.9 o 0.99 [8].

El algoritmo de momentum también ayuda a solucionar el problema de tener una matriz Hessiana mal condicionada. En la figura 17 tenemos como ejemplo un objetivo cuadrático que sufre de este inconveniente. Con rojo se encuentra el camino recorrido al utilizar momentum y con negro las direcciones de los gradientes. Se puede ver que momentum hace más eficiente la aproximación al mínimo, mientras que con SGD convencional se perdería tiempo en desplazamientos sobre direcciones que no contribuyen a disminuir el costo.

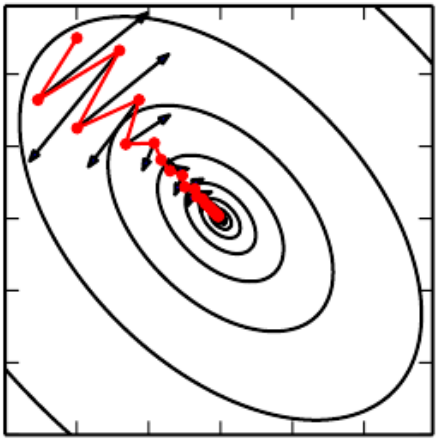


Figura 17: Momentum ayuda a optimizar objetivos con matrices Hessianas mal condicionadas.

Previamente, en SGD convencional, el tamaño del paso era la norma del gradiente multiplicada por la tasa de aprendizaje. Ahora el tamaño del paso depende de cuan grande y cuan alineada está la secuencia de gradientes. El tamaño del paso es más grande mientras más gradientes apunten sucesivamente en la misma dirección. Si durante la ejecución del algoritmo se observa siempre un gradiente g, entonces este va a acelerar en la dirección $-g$ hasta llegar a una velocidad terminal donde el tamaño de cada paso es $\eta \|g\| / (1 - \alpha)$. De acuerdo a esto, $\alpha = 0.9$ corresponde a una velocidad máxima de desplazamiento de 10 veces la que puede alcanzar el algoritmo de SGD.

A continuación mostramos la implementación de SGD+momentum para nuestra red:

```
def SGDMomentum(net, trng_set, val_set, loss_fn,
                epochs=30, bat_size=100, rate=0.001, momentum=0.9):
    x, y = trng_set
    trng_size = x.shape[1]
    vw, vb = [0], [0]
    for k in range(1, net.L):
        vw.append(np.zeros(net.w[k].shape))
        vb.append(np.zeros(net.b[k].shape))
    for n in range(epochs):
        shf_idx = np.random.permutation(trng_size)
        x, y = x[:, shf_idx], y[:, shf_idx]
```

```
        for k in range(0, trng_size, bat_size):
            net.backprop(x[:, k:k+bat_size], y[:, k:k+bat_size])
            for k in range(1, net.L):
                vw[k] = momentum * vw[k] - rate/bat_size * net.dw[k]
                vb[k] = momentum * vb[k] - rate/bat_size * net.db[k]
                net.w[k] =  net.w[k] + vw[k]
                net.b[k] =  net.b[k] + vb[k]
        acc, loss = performance(net, val_set, loss_fn)
        print('epoch: %2d | acc: %2.2f%% | loss: %.5f' % (n+1, acc*100, loss))
```

La diferencia con el algoritmo de SGD que habíamos implementado antes está en el paso de actualización y en la necesidad de definir e inicializar las variables que van a guardar las velocidades durante la ejecución.

4.3. Nesterov

La idea central detrás de este método es que cuando el vector actual de parámetros se encuentra en una posición θ, si solo miramos el término que contiene la inercia, podemos predecir que en principio el parámetro va a ser desplazado por αv. En consecuencia, podemos usar esta aproximación futura para calcular el gradiente, en vez de usar la posición actual. Esto en teoría posibilitaría un descenso un poco más "inteligente", ya que tenemos cierta noción de a donde estamos yendo antes de aplicar el gradiente. Por ejemplo, en SGD+momentum podría ocurrir que la inercia nos llevara a una colina, y que el gradiente evaluado en la posición actual también (con tasa de aprendizaje η), con lo cual terminaríamos yendo cuesta arriba. Pero si primero evaluamos el gradiente en la posición en la que nos está llevando la inercia, descubriríamos que estamos yendo cuesta arriba, y el gradiente apuntaría en la dirección de descenso, llevándonos de regreso cuesta abajo.

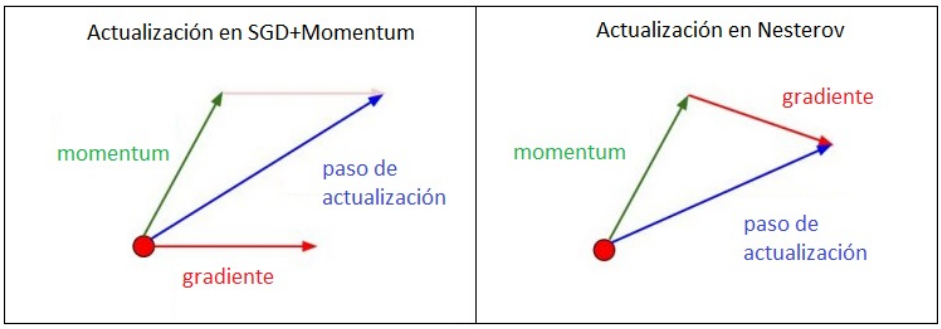


Figura 18: Comparación entre el paso de actualización de parámetros utilizando SGD + momentum y Nesterov.

Las fórmulas de actualización para el método de Nesterov son similares a las de

SGD+Momentum.

$$v \leftarrow \alpha v - \eta \nabla_\theta \left(\frac{1}{m} \sum_k^m L(f(x_k, \theta + \alpha v), y_k) \right), \tag{4.11}$$

$$\theta \leftarrow \theta + v. \tag{4.12}$$

Como se puede ver, la diferencia está en que el gradiente de la ecuación (4.11) se evalúa en $\theta + \alpha v$.

4.4. AdaGrad

Hasta ahora lo que hemos hecho en SGD y momentum ha sido realizar una actualización para todos los parámetros usando la misma tasa de aprendizaje. Específicamente, la actualización para cada parámetro en SGD es

$$\theta_i \leftarrow \theta_i - \eta g_i, \tag{4.13}$$

con $g = \nabla_\theta C$. En cambio, la regla de actualización de Adagrad [9] para cada parámetro es la siguiente:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{\sum_{k=0}^t g_{k,i}^2 + \epsilon}} g_{t,i}, \tag{4.14}$$

donde ϵ es un término que evita la división por cero, al que se le suele asignar un valor del orden de $1e{-}6$. Si denotamos con S_t el vector que contiene la suma del histórico de los gradientes para cada parámetro y hacemos $G_t = S_t \odot S_t$, entonces la regla de actualización en su forma vectorizada es

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t. \tag{4.15}$$

Lo que Adagrad hace es adaptar la tasa de aprendizaje de los parámetros, escalándolos de manera inversamente proporcional a la norma L^2 del histórico de los valores del gradiente, para cada parámetro. De esta manera, los parámetros con derivadas parciales más grandes tienen una disminución rápida en su tasa de aprendizaje, mientras que los parámetros con derivadas parciales más pequeñas tienen una disminución relativamente menor. El resultado de esto es que se logra un mayor progreso en las direcciones con pendientes más suaves y se evita el comportamiento zigzagueante que puede surgir si la matriz Hessiana no se encuentra bien condicionada.

El problema con Adagrad es la acumulación de los gradientes al cuadrado en el denominador durante el entrenamiento, haciendo que la tasa de aprendizaje disminuya hasta hacerse muy pequeña. En el contexto de optimización convexa, para el cual fue desarrollado el algoritmo, esta es una propiedad deseable, ya que permite que vaya desacelerando a medida que se va a acercando al mínimo global, de tal manera que quede en reposo una vez alcanzado. Sin embargo, en redes neuronales lo habitual es trabajar con optimización no convexa, y la antes mencionada acumulación de los gradientes puede prevenir el aprendizaje, haciendo que el algoritmo se quede atascado en un punto de inflexión, por ejemplo.

4.5. RMSProp

RMSProp [10] es una extensión de AdaGrad que busca solucionar el problema de la acumulación de gradientes, haciéndolo más apropiado en un contexto de optimización no convexa. Para eso lo que se hace es calcular una media móvil exponencial, definida de manera recursiva, donde la suma de todos los gradientes antes del paso t es decaída de acuerdo a un factor α, sumándole luego el gradiente actual, escalado apropiadamente. Si denotamos con $E\left[g^2\right]_t$ a la media móvil en el paso t, la formula sería la siguiente:

$$E\left[g^2\right]_t = \alpha E\left[g^2\right]_{t-1} + (1-\alpha)g_t^2. \tag{4.16}$$

Se suele asignar un valor de 0.9 para α, de manera similar a lo que se hace en SGD + momentum.

Si ahora recordamos que la formula de actualización para Adagrad es

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t, \tag{4.17}$$

lo único que tenemos que hacer es reemplazar G_t con $E\left[g^2\right]_t$, con lo cual obtenemos

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E\left[g^2\right]_t + \epsilon}} \odot g_t. \tag{4.18}$$

Lo que logramos con esto es que los gradientes de tiempos muy lejanos en la ejecución no tengan tanto peso al momento de actualizar los parámetros, evitando la disminución excesiva de la tasa de aprendizaje.

La cantidad $\sqrt{E\left[g^2\right]_t + \epsilon}$ se puede interpretar como una media cuadrática, abreviada en inglés como RMS (root mean square). Es por esta razón que el algoritmo recibe el nombre de RMSProp, que corresponde a Root Mean Square Propagation.

La rutina de actualización de los parámetros implementada para nuestra red es:

```
for k in range(0, trng_size, bat_size):
    net.backprop(x[:, k:k+bat_size], y[:, k:k+bat_size])
    for k in range(1, net.L):
        gw, gb = net.dw[k]/bat_size, net.db[k]/bat_size
        rw[k] = rho * rw[k] + (1 - rho) * gw**2
        rb[k] = rho * rb[k] + (1 - rho) * gb**2
        net.w[k] =  net.w[k] - rate/(np.sqrt(rw[k]) + 1e-7) * gw
        net.b[k] =  net.b[k] - rate/(np.sqrt(rb[k]) + 1e-7) * gb
```

Salvo el paso de actualización, el resto de la implementación es igual a la de SGD + momentum, por lo que la omitimos.

4.6. Adam

Adam [11] (Adaptive Moment Estimation) es otro método que calcula tasas de aprendizaje que se adaptan para cada parámetro. En adición a guardar una media móvil exponencial de los gradientes al cuadrado, este algoritmo también guarda una media móvil de

los gradientes (sin el cuadrado). Los promedios de los gradientes y sus cuadrados tienen entonces las formulas

$$m_t = \alpha m_{t-1} + (1-\alpha) g_t, \tag{4.19}$$
$$v_t = \beta v_{t-1} + (1-\beta) g_t^2. \tag{4.20}$$

Se puede pensar m_t como un estimador del primer momento de los gradientes, mientras que v_t es un estimador del segundo momento; de ahí viene el nombre del método. Es en este sentido que Adam es similar a una combinación de RMSProp y momentum.

Dado que m_t y v_t son inicializados como vectores cero, esto ocasiona que estén sesgados hacia el cero, particularmente durante los pasos iniciales, cuando las tasas de decaimiento son pequeñas. Para contrarrestar esto, se introducen unos estimadores corregidos:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1-\alpha^t}, \\ \hat{v}_t &= \frac{v_t}{1-\beta^t}. \end{aligned} \tag{4.21}$$

Estos son los que luego se usan para actualizar los parámetros, de la misma manera que en RMSProp:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t. \tag{4.22}$$

Los valores recomendados para α, β y ϵ suelen ser 0.9, 0.999 y 10^{-8} respectivamente.

5. Redes Neuronales Convolucionales

5.1. Motivación

Nuestra implementación de la red neuronal para clasificar los dígitos en MNIST ha funcionado de manera respetable hasta ahora. Esta red neuronal está basada en una arquitectura de capas completamente conexas, lo cual significa que cada unidad toma información de todas las unidades de la capa previa. Si pensamos en la relación entre la capa de entrada y la primera capa oculta, lo que esto nos dice es que la capa oculta toma en cuenta todos los píxeles de la imagen a la hora de evaluar la entrada. Esto no es del todo conveniente en nuestro caso, ya que en una imagen esperamos que cada píxel esté relacionado de alguna manera con sus vecinos más próximos, pero no con aquellos que se encuentran más lejos, por lo que no necesariamente tiene sentido que cada unidad procese todos los píxeles, sino que en todo caso debería limitarse a una ventana dentro de la imagen, la cual se denomina *campo receptivo local*.

Para ver esto, pensemos la imagen como una grilla de píxeles, que en nuestro caso sería de 28×28, con valores que corresponden a la intensidad de cada píxel. Cada neurona de la capa oculta va a procesar una región de esa imagen, que para este ejemplo será de 5×5, por lo que las conexiones se verían como en la figura 19. Por cada región de 5×5 de la imagen va a corresponder una neurona oculta que la procesa, de modo que si desplazamos el campo receptivo local por toda la imagen obtenemos un mapeo como el de la figura 20.

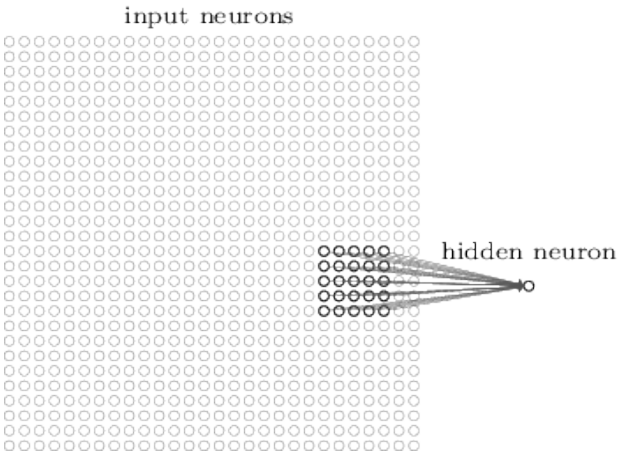


Figura 19: Conexión entre una neurona y su campo receptivo local.

La expresión que describe la salida de cada neurona para nuestro ejemplo es

$$a_{j,k} = \phi\left(b + \sum_{m,n} w_{m,n} x_{j+m,k+n}\right), \tag{5.1}$$

donde w es la matriz de pesos, b es el sesgo, y x es la entrada. Esta expresión es en esencia una convolución, por lo que al conjunto de parámetros compartidos w y b se les suele llamar *filtro* o *kernel*. Utilizando un solo filtro se detecta solo una característica, siendo habitual combinar un conjunto de varios filtros por capa, para así poder extraer

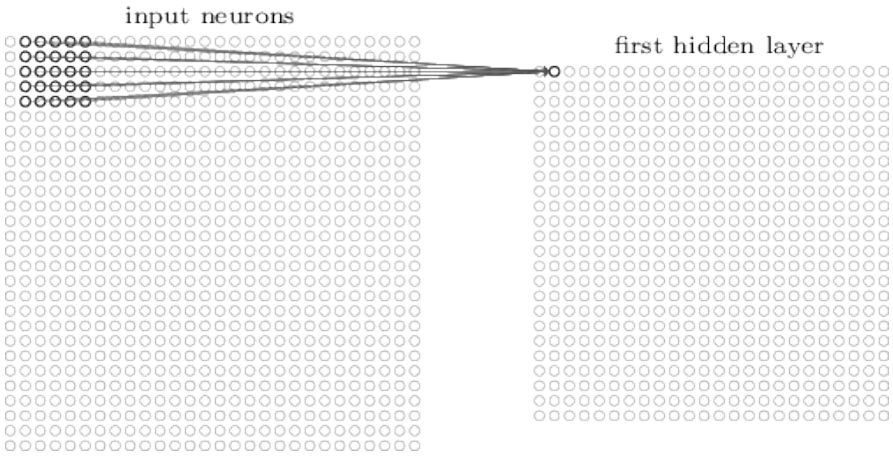


Figura 20: Mapa de características.

varias características de la imagen. En la figura 21 se pueden ver varios filtros de 5×5. Los píxeles más oscuros corresponden a pesos más grandes y los más claros a pesos más pequeños o incluso negativos. En consecuencia, cada filtro va a responder más a aquellos píxeles de la entrada que se corresponden con las zonas más oscuras, por lo que de alguna manera la imagen del filtro nos dice a que característica es sensible.

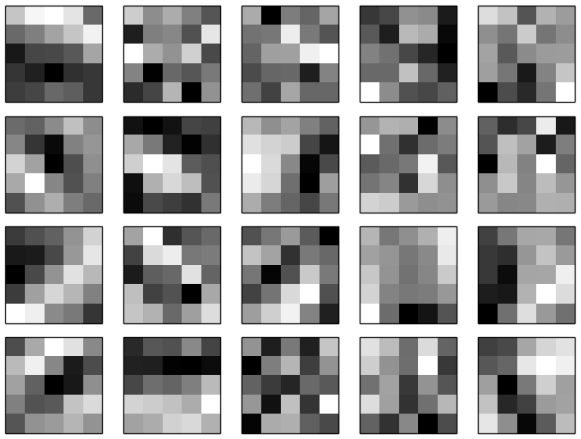


Figura 21: Imágenes de varios filtros.

Al igual que lo hecho hasta ahora, la neurona cuenta con un sesgo, y para cada conexión corresponde un peso, los cuales son parámetros que deben ser aprendidos. No obstante, una particularidad en redes convolucionales es que estos parámetros son compartidos. Todas las neuronas usan los mismos pesos y sesgos, lo único que cambia es la región de la imagen que procesan. Las motivaciones detrás de esto son múltiples. Por un lado, lo que uno logra con esto es que todas las neuronas identifiquen la misma característica en la imagen, solo que en distintas ubicaciones. Esto tiene sentido ya que si los parámetros se configuran de manera tal que uno pueda detectar, por ejemplo, una región con un gato, esta habilidad es igualmente útil en cualquier lugar de la imagen. Por otra parte, esta compartición permite disminuir la cantidad de parámetros de manera substancial, ahorrando la cantidad de procesamiento necesario por capa, al igual que los

requerimientos de memoria. Supongamos que tenemos m entradas y n salidas. En ese caso la multiplicación matricial requiere $m \times n$ parámetros y el algoritmo requiere un tiempo de ejecución de $O(m \times n)$ por cada ejemplo. Si limitamos el número de conexiones que cada salida puede tener a k, entonces solo requerimos $k \times n$ parámetros y $O(k \times n)$ tiempo de ejecución. Muchas veces se puede obtener un rendimiento aceptable haciendo k varios ordenes de magnitud más pequeño que m. En el caso de nuestro filtro de 5×5, este conecta la salida con 25 elementos de la entrada, de un total de 784. Esto representa una diferencia de un orden de magnitud. Para imágenes de tamaños a los que estamos acostumbrados hoy en día, como puede ser 1280×720, ya hablamos de diferencias de 4 ordenes de magnitud. En lo que respecta al ahorro de memoria, el razonamiento es análogo.

Otra propiedad que se deriva de la compartición de parámetros en redes convolucionales es la *equivarianza* a la traslación. Ser equivariante significa que si la entrada cambia entonces la salida cambia de la misma manera. Específicamente, una función f es equivariante a otra función g si $f(g(x)) = g(f(x))$. En el caso de la convolución, si hacemos que g sea cualquier función que traslada la entrada, entonces la convolución es equivariante a g. Recordemos que al aplicar la convolución a una imagen se crea un mapa de la ubicación de ciertas características en la entrada. Si movemos el objeto en la entrada la equivarianza nos garantiza que esta representación se mueve de la misma manera en la salida. Esto es útil cuando sabemos que alguna función de un número pequeño de píxeles vecinos es útil aplicada a varias regiones de la imagen, como en el ejemplo del gato que mencionamos antes.

5.2. Convolución

Supongamos que estamos monitoreando la ubicación de un vehículo con un sensor. Este sensor nos provee una salida individual $x(t)$, la posición del vehículo en el tiempo t. Tanto x como t toman valores reales, por lo que podemos obtener una lectura diferente del sensor en cualquier instante de tiempo.

Supongamos ahora que nuestro sensor tiene ruido. Para obtener una estimación con menos ruido uno podría promediar varias mediciones. Las mediciones más recientes son más relevantes, por lo que nos va a interesar trabajar con un promedio pesado que les asigne un mayor peso a las mismas. Para eso introducimos una función $w(a)$, donde a es la edad de la medición. Aplicando el promedio pesado en cada momento t obtenemos una función que nos provee una estimación suavizada de la posición del vehículo:

$$s(t) = \int x(a)w(t-a)\,da. \tag{5.2}$$

Esta operación recibe el nombre de convolución y se suele denotar como

$$s(t) = (x * w)(t), \tag{5.3}$$

donde x es la entrada y w es el *kernel*. A la salida se la suele denominar *mapa de características*.

En nuestro ejemplo necesitamos que w sea una distribución de probabilidad válida, ya que sino la salida no será un promedio pesado. También necesitamos que w sea 0 para

todos los argumentos negativos, porque sino va a estar mirando al futuro. No obstante, esas limitaciones son específicas de este ejemplo.

Muchas veces, la idea de tener mediciones en cada instante no es realista. Al trabajar con información en una computadora el tiempo se encuentra discretizado, por lo que es posible que el sensor de nuestro ejemplo solo nos provea información a intervalos regulares. En ese caso el tiempo queda indexado por un valor entero t y la convolución discreta queda definida como

$$s(t) = (x * w)(t) = \sum_{-\infty}^{\infty} x(a)w(t-a). \tag{5.4}$$

Por otra parte, generalmente la entrada y el número de parámetros son finitos, por lo que en ese caso la sumatoria anterior puede ser implementada como una suma finita.

En imágenes nos va a interesa usar convolución sobre más de un eje. Si tenemos una imagen I y un kernel K de dos dimensiones, el análogo de (5.4) es

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n). \tag{5.5}$$

La convolución es conmutativa, por lo que también podemos escribirla como

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n), \tag{5.6}$$

expresión que a veces resulta más conveniente a la hora de implementarla.

Si comparamos la ecuación (5.5) y (5.1) (descartando el sesgo), vemos que hay una discrepancia en los signos, pero las operaciones son muy similares. La operación que en realidad nos interesa en nuestro caso es la presente en (5.1), que se denomina correlación cruzada:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n). \tag{5.7}$$

La diferencia entre la convolución y la correlación cruzada es que en la convolución uno "gira" el kernel, de tal manera que a medida que m y n aumentan, el índice en la entrada también aumenta, pero el del kernel disminuye. Esto permite obtener la propiedad conmutativa que mencionamos antes. Esta propiedad no nos interesa tanto en nuestras aplicaciones, siendo la correlación cruzada la que mejor describe la operación que queremos aplicar. La mayoría de las bibliotecas utilizadas en redes neuronales implementan la correlación cruzada, pese a que luego le llamen convolución.

La operación de convolución utilizada en la práctica también difiere de otras maneras respecto a la definición habitualmente utilizada en matemática. En el contexto de redes neuronales, cuando nos referimos a convolución se está pensando en la aplicación de varias convoluciones en paralelo. Ya vimos antes que cada filtro es capaz de extraer solo una característica, de modo que lo que nos interesa es combinar varios de estos filtros. Es por esto que es conveniente pensar al filtro como una sola entidad de $p \times q \times r$, donde p y q representan las filas y columnas del filtro y r el número de filtros que estamos utilizando. Lo que tenemos entonces es que nuestro filtro es un tensor de dimensión tres. Por otra parte, lo dicho antes corresponde al caso de que la entrada sea 2-D; no obstante, nuestra entrada tampoco se suele limitar a una grilla de valores reales, sino que suele contar con varios canales. Por ejemplo, en el caso de una imagen a color, no tenemos solo una matriz

con las intensidades presentes en cada píxel, sino que tenemos tres de estas imágenes, una por cada canal de color (rojo, verde, azul). De esta manera, se puede pensar a la imagen como un tensor 3-D, y en consecuencia nuestro filtro pasa a ser 4-D, ya que se necesita un filtro distinto por cada canal de entrada.

Vamos a asumir entonces que tenemos como filtro un tensor 4-D al que denotamos por W, con elementos $W_{i,j,k,l}$ que nos dan los pesos que corresponden a la conexión entre una unidad en el canal i de la salida y una unidad en el canal j de la entrada, con un desplazamiento de k filas y l columnas entre la unidad de salida y la unidad de entrada. La entrada consiste de un tensor X con elementos $X_{i,j,k}$ que nos dan el valor de la entrada en el canal i, fila j y columna k, y asumimos que la salida Z tiene la misma forma. La operación de convolución que nos interesa utilizar en la aplicación de redes neuronales convolucionales para imágenes, toma entonces la forma

$$Z_{i,j,k} = \sum_{l,m,n} X_{i,l,j+m,k+n} W_{i,l,m,n}, \tag{5.8}$$

donde estamos indexando empezando desde cero (en caso contrario aparecería un -1).

Algo que nos puede interesar al convolucionar es mover el filtro con un paso mayor a uno, con lo cual logramos una salida más pequeña, en lo que se conoce como *downsampling*. Para lograr esto se puede adaptar la formula (5.8), quedando

$$Z_{i,j,k} = \sum_{l,m,n} X_{i,l,j\cdot s+m,k\cdot s+n} W_{i,l,m,n}, \tag{5.9}$$

donde s es el paso de la convolución. También se puede definir un paso diferente para cada dirección del movimiento.

Una característica esencial de cualquier implementación de red convolucional es la habilidad de rellenar con ceros la entrada X. Esto se denomina *padding*. Si hacemos que I sea el tamaño de la entrada, F el tamaño del filtro, S el paso de la convolución, y P la cantidad de padding utilizada, la fórmula que calcula el tamaño de la salida es $(I - F + 2P)/S + 1$. Para dar un ejemplo, supongamos que la entrada es de 7×7, el filtro es de 3×3, el paso es 1 y el relleno es 0. En ese caso, la salida va a tener un tamaño de 5×5. Muchas veces puede interesarnos mantener el tamaño de la salida igual al de la entrada, o mantenerlo controlado de alguna otra manera, por lo que en ese caso es necesario ajustar el padding de manera conveniente.

Hay tres casos de padding que se suelen trabajar. El primero es no utilizar padding, permitiendo que el filtro visite solamente aquellas posiciones para las cuales está contenido completamente en la entrada. Al trabajar con MATLAB y otras bibliotecas numéricas, esto suele denominarse convolución *válida*. En este caso todos los píxeles de la salida son una función del mismo número de píxeles de la entrada, por lo que el comportamiento de un píxel de salida será un poco más regular. De acuerdo a la formula del párrafo anterior, la salida en este caso será de ancho $I - F + 1$. Otro caso es aquel en el que se mantiene el tamaño de la salida igual al de la entrada (convolución *"same"* en MATLAB), lo cual le permite a la red contener tantas capas convolucionales como pueda soportar el hardware, ya que no hay encogimiento de la información a medida que va pasando por las diferentes capas. La desventaja es que los píxeles del borde tienen influencia sobre menos píxeles de la salida que los del centro. Esto puede hacer que los píxeles del borde

sean subrepresentados. La otra posibilidad es utilizar una convolución *completa*, en la que se agregan suficientes ceros para que cada píxel de la entrada sea visitado F veces en cada dirección, resultando en una salida de ancho $I + F - 1$. En este caso los píxeles del borde de la salida son una función de menos píxeles que aquellos que se encuentran en el centro, haciendo difícil aprender un solo filtro que se comporte correctamente en todas las posiciones del mapa de características.

En algunos casos uno puede querer realizar una operación similar a la convolución pero sin compartir los parámetros, con lo cual uno estaría trabajando con capas localmente conexas. En ese caso, los índices al tensor de pesos son los siguientes: i para el canal de salida, j para la fila de salida, k para la columna de salida, l para el canal de entrada, m para el desplazamiento en las filas dentro de la entrada y n para el desplazamiento en las columnas de la entrada. La fórmula para calcular la salida lineal es

$$Z_{i,j,k} = \sum_{l,m,n} X(l, j+m, k+n) W(i, j, k, l, m, n). \tag{5.10}$$

Las capas localmente conexas son útiles cuando sabemos que cada característica debería ser una función de una parte pequeña del espacio, pero no hay razones para creer que estas se repitan en todo el espacio. Por ejemplo, si consideramos el caso de querer de decidir si una imagen es una cara o no, vemos que solo necesitamos buscar la imagen de la nariz en el centro de la imagen.

5.3. Pooling

Una capa convolucional típica consiste de tres etapas. En la primera, la capa realiza varias convoluciones en paralelo para producir un conjunto de activaciones lineales. En la segunda etapa, cada activación lineal es pasada por una función de activación no lineal, como puede ser la ReLU. En la tercer etapa se utiliza una función de *pooling* para simplificar la información de la capa de salida.

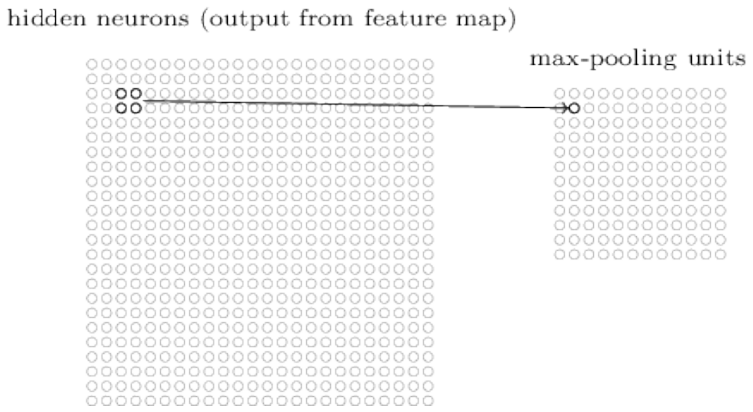


Figura 22: Max pooling

Lo que la capa de pooling hace es tomar el mapa de características de la capa convolucional y preparar una versión condensada. Por ejemplo, cada unidad de la capa de pooling puede resumir una región de 2×2 de la capa previa tomando el máximo de la región. Esto se conoce como *max-pooling*. Otras posibilidades incluyen tomar la norma

L^2 o un promedio pesado basado en la distancia al píxel central. El uso de las distintas técnicas depende de la naturaleza del problema.

Como mencionamos antes, la capa convolucional suele involucrar varios filtros, obteniendo varios mapas de características. Max-pooling se aplica a cada mapa por separado, por lo que si tenemos tres mapas de características, vamos a tener una capa de pooling de tres canales, como en la figura 23.

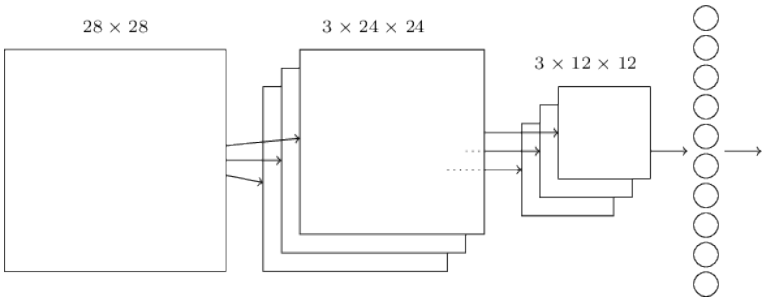


Figura 23: Capa convolucional de 3 filtros seguida de capa max-pooling de 3 canales.

Lo que pooling nos permite es hacer la representación aproximadamente invariante frente a traslaciones pequeñas de la entrada. Esto significa que si trasladamos la entrada solo un poco, entonces la salida de las unidades de pooling no cambia. Esta invarianza a traslaciones locales es una propiedad útil cuando lo que nos importa es si una característica se encuentra presente, y no tanto su ubicación exacta. Por ejemplo, al determinar si una imagen contiene una cara, no necesitamos saber la ubicación de los ojos con una precisión a la escala del píxel, sino que solo nos interesa saber si hay un ojo a la izquierda y otro a la derecha de la cara.

Dado que pooling resume las respuestas sobre toda una región, es posible usar menos unidades de pooling que unidades detectoras, espaciando las regiones k píxeles. De esta manera se aumenta la eficiencia computacional ya que la próxima capa va a tener k veces menos entradas que procesar.

Otra utilidad de la capa de pooling es que nos permite trabajar con entradas de tamaño variable. Por citar un caso, uno puede tener imágenes de tamaño variable, pero la entrada a la capa de clasificación debe tener un tamaño fijo. Esto se suele lograr variando las dimensiones de las regiones de pooling de tal manera que la capa de clasificación reciba siempre una entrada del mismo tamaño. Por ejemplo, la capa final de pooling puede ser definida de tal manera que devuelva cuatro elementos, uno por cada cuadrante de la imagen.

5.4. Implementación de la Red Convolucional

Para la implementación de la red convolucional vamos a utilizar una biblioteca de deep learning llamada PyTorch. PyTorch nos provee en un formato accesible muchas de las funcionalidades usualmente utilizadas al trabajar con redes neuronales. Por ejemplo, nos provee de operaciones de convolución específicamente diseñadas para este contexto, simplificando enormemente la tarea de implementar una red convolucional. No obstante, no solo está la conveniencia de tener una biblioteca que empaqueta muchas de las funciones que necesitamos, sino que PyTorch trae a la mesa dos funcionalidades esenciales

para desarrollar redes neuronales. Una de ellas es la capacidad de trabajar con la GPU de forma fluida, aumentando enormemente la velocidad de entrenamiento, cosa que resulta muy importante si se está trabajando con arquitecturas con muchas capas, como las que se han usado en redes más recientes. Otra característica fundamental de estas bibliotecas es que nos proveen de métodos para realizar backpropagation de manera automática, basándose en el gráfico computacional de la red. De esta manera no es necesario especificar manualmente cada paso de backpropagation, sino que solo debemos definir como fluye la información hacia adelante y del resto se ocupa PyTorch. Lo que esto nos permite es poder trabajar con un solo árbol computacional, en vez de tener que preocuparnos por dos. Cualquier cambio que hagamos en la arquitectura de la red se ve reflejado en los gradientes que obtenemos luego.

La red convoluciónal que vamos a implementar está basada en la Lenet-5 [12], siendo una de las primeras que se desarrollaron y teniendo mucho éxito en el problema para el cual fue diseñada, que es justamente la clasificación de dígitos. Esta red se basa en una combinación de dos capas Conv+Pool, seguidas luego de tres capas totalmente conexas, por lo que se puede ver como una expansión de la red que habíamos trabajado hasta ahora, añadiéndole el componente convolucional al principio. Todas las unidades de las capas ocultas van a ser ReLU, mientras que la salida que utilizaremos será la softmax.

La arquitectura Lenet-5 comienza por aplicarle un padding a las imágenes de 28×28 de MNIST para que resulten de 32×32. Esto se hace para que al aplicarle la convolución obtengamos una salida del mismo tamaño que la entrada original, es decir, de 28×28. En terminología MATLAB lo que estamos haciendo es una convolución "same". La primer capa convolucional va a aplicar 6 filtros de 5×5, por lo que en principio estamos extrayendo 6 características de la imagen. Los mapas de características resultantes son pasados luego a una capa de pooling con paso 2, resumiendo la información en 6 mapas de características de 14×14, la mitad del tamaño. En la segunda capa convolucional se repite el proceso, pero aplicando 16 filtros de $6 \times 5 \times 5$ sobre la salida de la capa anterior, sin aplicar padding. Esto nos da como resultado 16 mapas de características de 10×10, a los cuales se les vuelve a aplicar pooling con paso 2, condensando la información en 16 mapas de 5×5. Toda esta información la podemos representar por un vector de longitud $16 \times 5 \times 5$ que luego es procesado por una serie de capas completamente conexas para así obtener una salida de tamaño 10.

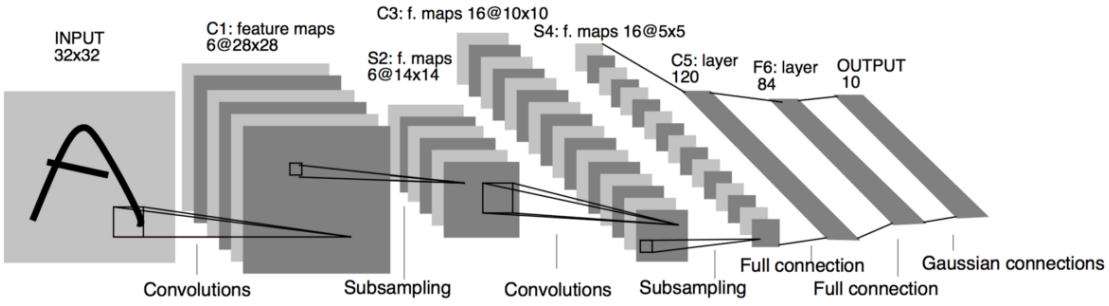


Figura 24: Lenet-5.

Como hicimos con la primera red que implementamos, comenzamos por importar las herramientas que necesitamos. Esto se hace en las siguientes lineas:

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import torch.utils.data as data
import torchvision.datasets as datasets
import torchvision.transforms as tr
```

Las primeras tres lineas importan la biblioteca PyTorch y algunas utilidades que usaremos para construir la red. La cuarta linea importa el módulo que nos provee los optimizadores para el entrenamiento. Las restantes nos proveen de los conjuntos de datos y las herramientas para procesarlos.

Empezamos ahora con la definición de la red, que se hace de una manera similar a lo hecho hasta ahora, solo que omitiendo la definición del método de backprop. El código es como sigue:

```
#%% Arquitectura de la red

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = F.max_pool2d(torch.relu(self.conv1(x)), 2)
        x = F.max_pool2d(torch.relu(self.conv2(x)), 2)
        x = x.view(-1, 16 * 5 * 5)
        x = torch.relu(self.fc1(x))
        x = torch.relu(self.fc2(x))
        x = torch.log_softmax(self.fc3(x), dim=1)
        return x
```

En la inicialización definimos las capas que utilizaremos. Primero tenemos una capa convolucional que tiene un canal de entrada y seis de salida, utilizando 6 filtros de 5×5 para realizar la convolución. Luego definimos la otra capa convolucional, que tiene 6 canales de entrada y 16 de salida, y sigue utilizando un filtro de 5×5. A esto le sigue la definición de la primer capa "fully-connected" (FC), que va a procesar la salida de la segunda capa después de haberle aplicado pooling, por lo que recibe un vector de 16*5*5, para luego devolver un vector de 120 elementos. De ahí pasamos a la siguiente capa, que toma los 120 y devuelve 84, y por último la tercer capa toma los 84 y aplica un último procesamiento para devolver la salida de 10 elementos que representan las probabilidades.

El segundo método nos dice como se propaga la información hacia adelante en la red. La variable x representa la entrada y se puede ver que cada linea va modificando x hasta devolver la salida. La primer linea de la función `forward` aplica la primer capa

convolucional a la entrada, luego la pasa por la función de activación, que en nuestro caso es la ReLU, y por último resume la información por medio de la operación de pooling. Todo esto podemos pensarlo como una única operación Conv+Relu+Pool. Esto se vuelve a repetir en la segunda linea, pero utilizando la capa convolucional número dos. La tercer linea vectoriza la salida, para poder procesarla con las capas FC. Esto se hace en las líneas 4 y 5, con una combinación FC+ReLU. En la línea 6 se procesa la información por medio de la última capa FC y se le aplica la función Softmax para obtener la salida.

Con eso tenemos definida la red, por lo que ahora necesitamos cargar los datos. Esto se hace con el siguiente código.

```
#%% Carga de datos

transform = tr.Compose([tr.Pad(2), tr.ToTensor()])
dataset = datasets.MNIST('./data', transform=transform)
trainset, valset, _ = data.random_split(dataset, (50000, 10000, 0))
testset = datasets.MNIST('./data', transform=transform, train=False)

train_loader = data.DataLoader(trainset, batch_size=32, shuffle=True)
val_loader = data.DataLoader(valset, batch_size=10000, shuffle=False)
test_loader = data.DataLoader(testset, batch_size=10000, shuffle=False)

val_data = iter(val_loader)
val_images, val_labels = val_data.next()
```

Lo primero que hacemos es definir una transformación que se ocupa de aplicarle un padding de 2 a cada ejemplo del conjunto de datos y luego transformarlo en tensor, para así poder trabajarlo. Luego cargamos el conjunto de entrenamiento MNIST y le aplicamos la transformación anterior. Este conjunto luego es dividido en un conjunto de entrenamiento y otro de validación, de 50000 y 10000 elementos respectivamente. También cargamos el conjunto de prueba, aunque por ahora no lo usaremos. En las últimas tres lineas lo que hacemos es generar unos cargadores de datos, que se encargan de alimentar el algoritmo de optimización durante el entrenamiento. Configuramos el tamaño del lote para el conjunto de entrenamiento en 32, como hemos estado haciendo hasta ahora, mientras que para el conjunto de validación queremos el lote completo, ya que lo usamos para monitorear el entrenamiento.

A continuación se crea la red, se define la función de pérdida, que en este caso será el costo NLL y luego se genera el optimizador, que por ahora seguirá siendo SGD.

```
#%% Construcción de la red, función pérdida, y optimizador.

net = Net()
loss_fn = nn.NLLLoss()
optimizer = optim.SGD(net.parameters(), lr=1e-1, weight_decay=0)
n_epochs = 20
```

Al optimizador lo configuramos con una tasa de aprendizaje de 0.1 y sin decaimiento de pesos, como hemos venido haciendo hasta ahora. Al número de épocas lo definimos en 20. Ahora lo que sigue es el bloque de entrenamiento:

```
#%% Entrenamiento
for epoch in range(n_epochs):
    for images, labels in train_loader:
        outputs = net(images)
        loss = loss_fn(outputs, labels)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    # Calculo del rendimiento por época
    outputs = net(val_images)
    test_loss = loss_fn(outputs, val_labels)
    pred = torch.argmax(outputs, dim=1)
    accuracy = torch.sum(pred == val_labels).item()/len(val_labels)

    # Resultados
    print('epoch: %2d | trng loss: %.3f | test loss: %.3f | acc: %.3f'\
          % (epoch+1, loss.item(), test_loss.item(), accuracy))
```

Durante el entrenamiento iteramos tantas veces como número de épocas. En cada época lo que hacemos es ir cargando sucesivamente lotes del conjunto de entrenamiento, separándolos en imágenes y rótulos. Las imágenes son procesadas por la red para obtener una salida y esta salida se usa junto con los rótulos para calcular la pérdida. Esto constituiría el paso de feedforward. A continuación se limpian los gradientes y se hace el paso de backpropagation con `loss.backward()`. Una vez obtenidos los gradientes se le pide al optimizador que haga un paso de actualización con `optimizer.step()`. Después de haber pasado por todos los elementos del conjunto de entrenamiento, se calcula el rendimiento y se muestran los resultados concluyendo la época.

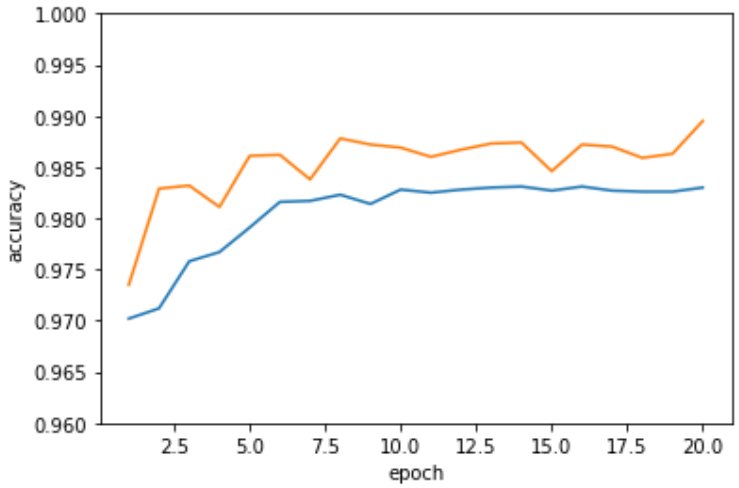


Figura 25: Comparación del rendimiento entre la red convolucional (naranja) y la red totalmente conexa (azul).

Si ejecutamos el programa, obtenemos los siguientes resultados:

```
epoch:  1 | trng loss: 0.002 | test loss: 0.075 | acc: 0.976
epoch:  2 | trng loss: 0.009 | test loss: 0.052 | acc: 0.985
epoch:  3 | trng loss: 0.210 | test loss: 0.044 | acc: 0.987
epoch:  4 | trng loss: 0.105 | test loss: 0.058 | acc: 0.982
epoch:  5 | trng loss: 0.000 | test loss: 0.045 | acc: 0.987
...
epoch: 16 | trng loss: 0.000 | test loss: 0.044 | acc: 0.989
epoch: 17 | trng loss: 0.010 | test loss: 0.053 | acc: 0.988
epoch: 18 | trng loss: 0.001 | test loss: 0.046 | acc: 0.989
epoch: 19 | trng loss: 0.000 | test loss: 0.048 | acc: 0.990
epoch: 20 | trng loss: 0.001 | test loss: 0.056 | acc: 0.989
```

Hemos logrado finalmente una exactitud del 99 %, superando así la barrera que nos detenía con la arquitectura anterior basada solo en capas FC.

Conclusiones

Nuestra primera implementación del perceptrón multicapa mostró un rendimiento inicial aceptable para el problema de clasificación de dígitos escritos a mano, pero también mostró cierta sensibilidad a las condiciones iniciales y lentitud durante el proceso de aprendizaje. En este contexto, el reemplazo de la función costo cuadrático por la entropía cruzada nos brindó una mejora importante en la velocidad del entrenamiento. Otro aspecto que probó ser muy importante para mejorar el aprendizaje de la red fue la elección de un método apropiado para inicializar los parámetros. Esto, combinado con la utilización de las unidades tanh, nos permitió obtener resultados muy superiores a los iniciales, tanto en la velocidad del entrenamiento como en la exactitud final, superior al 98 %. En lo que respecta a la elección de las unidades ocultas y de salida, también se examinaron otras posibilidades, como son las unidades ReLU y softmax, pero no se observaron mejoras substanciales para nuestro problema.

Finalmente se logró una mejora notable en el rendimiento al cambiar la arquitectura, empleando una red neuronal convolucional. Este tipo de redes funcionan particularmente bien cuando son aplicadas a imágenes, ya que su arquitectura tiene en cuenta la estructura espacial de los datos que se procesan. El resultado fue una exactitud del 99 %, superando las barreras que nos detenían con la arquitectura utilizada previamente. Incluso es factible mejorar estos resultados aún más combinando técnicas de regularización como dropout y aumento de datos, las cuales han sido utilizadas con éxito para obtener exactitudes superiores al 99.5 %.

En la actualidad las redes convolucionales se utilizan ampliamente en el campo de visión artificial, aplicadas a tareas como clasificación, reconocimiento de objetos en imágenes y subtitulado de imágenes. Las arquitecturas se han vuelto cada vez más profundas, con algunas de las más recientes superando las 100 capas. El aumento de la profundidad, junto con algunas mejoras sobre la red convolucional básica, han permitido atacar problemas cada vez más complejos. En este sentido, lo logrado con el conjunto MNIST representa apenas la punta del iceberg en términos de las capacidades de este tipo de esquemas. Uno de los desafíos más populares hoy en día en el reconocimiento visual de objetos es el provisto por ImageNet. Este conjunto contiene más de 14 millones de imágenes de diversos objetos y 20000 categorías con las que clasificarlos. En 2012 se dio un paso importante en este problema con la introducción de la AlexNet, que logró reducir el error al 15 %. Desde entonces se han logrado mejoras cada año, obteniendo desde el 2017 una exactitud por encima del 95 %, e incluso esto no es más que solo un pequeño vistazo al panorama actual.

Más allá de las redes neuronales convolucionales existen otras arquitecturas que no han sido vistas en este trabajo, como pueden ser las redes neuronales recursivas o los autoencoders, por mencionar algunas. Estas se aplican en diversos campos, como puede ser en el reconocimiento de voz, el diagnóstico de enfermedades, la predicción de tendencias, el catalogado de contenido, la publicidad dirigida, etc. Se puede apreciar que el panorama de aplicaciones para las redes neuronales es amplio y variado, y este trabajo no es más que una mirada sobre algunas de las ideas sobre las que se funda el campo de deep learning, sirviendo de base para la exploración de técnicas más avanzadas que permitan la resolución de problemas más complejos.

A. Implementación 1

```
import numpy as np
import numpy.random as rnd

#%% Arquitectura de la red

neurons = [784, 400, 100, 10]
L = len(neurons)
w, b, dw, db, a, z = [L*[0] for k in range(6)]

def feedforward(x):
    a[0] = x
    z[0] = x
    for k in range(1, L):
        z[k] = w[k] @ a[k-1] + b[k]
        a[k] = sigmoid(z[k])
    return a[-1]

def backprop(x, y):
    feedforward(x)
    delta = (a[-1] - y) * sigmoid_prime(z[-1])
    dw[-1] = delta @ a[-2].T
    db[-1] = np.sum(delta, axis=1, keepdims=True)
    for k in reversed(range(1, L-1)):
        delta = (w[k+1].T @ delta) * sigmoid_prime(z[k])
        dw[k] = delta @ a[k-1].T
        db[k] = np.sum(delta, axis=1, keepdims=True)

#%% Inicializaciones

def xavier_init():
    for k in range(1, L):
        in_size = neurons[k-1]
        out_size = neurons[k]
        w[k] = np.sqrt(2/(in_size+out_size)) * rnd.randn(out_size, in_size)
        b[k] = rnd.randn(out_size, 1)

def lecun_init():
    for k in range(1, L):
        in_size = neurons[k-1]
        out_size = neurons[k]
        w[k] = np.sqrt(1/in_size) * rnd.randn(out_size, in_size)
        b[k] = rnd.randn(out_size, 1)

def normal_init():
    for k in range(1, L):
        in_size = neurons[k-1]
```

```
        out_size = neurons[k]
        w[k] = rnd.randn(out_size, in_size)
        b[k] = rnd.randn(out_size, 1)

def zero_init():
    for k in range(1, L):
        in_size = neurons[k-1]
        out_size = neurons[k]
        w[k] = rnd.zeros(out_size, in_size)
        b[k] = rnd.zeros(out_size, 1)


#%% Método de entrenamiento

def SGD(trng_set, val_set, loss_fn, epochs=30, bat_size=100, eta=1):
    x, y = trng_set
    trng_size = x.shape[1]
    for n in range(epochs):
        shf_idx = rnd.permutation(trng_size)
        x, y = x[:, shf_idx], y[:, shf_idx]
        for k in range(0, trng_size, bat_size):
            backprop(x[:, k:k+bat_size], y[:, k:k+bat_size])
            for k in range(1, L):
                w[k] =  w[k] - eta/bat_size * dw[k]
                b[k] =  b[k] - eta/bat_size * db[k]
        acc, loss = performance(val_set, loss_fn)
        print('epoch: %2d | acc: %2.2f%% | loss: %.5f' % (n+1, acc*100, loss))

#%% Monitoreo del rendimiento

def performance(test_set, loss_fn):
    x, y = test_set
    test_size = x.shape[1]
    feedforward(x)
    pred = np.argmax(a[-1], axis=0)
    label = np.argmax(y, axis=0)
    acc = np.mean(pred == label)
    loss = loss_fn(a[-1], y) / test_size
    return acc, loss

#%% Costo

def quad_cost(a, y):
    return 1/2 * np.sum((a - y)**2)

def quad_grad(a, y):
    return a - y
```

```
def cross_entropy_cost(a, y):
    return -np.sum(y*np.log(a) + (1 - y)*np.log(1 - a))

#%% Activacion

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_prime(x):
    s = sigmoid(x)
    return s * (1 - s)

#%% Carga de datos

def load_data(trng_size=50000, val_size=10000,
              shuffle=False, normalize=True):
    total_size = trng_size + val_size
    x, y = np.load('../data/trng_data.npy')
    if shuffle:
        shf_idx = np.random.permutation(x.shape[1])
        x, y = x[:, shf_idx], y[:, shf_idx]
    if normalize:
        x = (x - np.mean(x, axis=0))/np.std(x, axis=0)
    trng_set = (x[:,0:trng_size], y[:,0:trng_size])
    val_set = (x[:,trng_size:total_size], y[:,trng_size:total_size])
    return trng_set, val_set

#%% Main

trng_set, val_set = load_data()
normal_init()
SGD(trng_set, val_set, quad_cost, epochs=30, bat_size=32, eta=1)
```

B. Implementación 2

```
import numpy as np
import numpy.random as rnd

#%% Arquitectura de la red

class Net:
    def __init__(self, neurons):
        self.neurons = neurons
        self.L = len(neurons)
        self.w, self.b, self.dw, self.db, self.a, self.z = \
        [self.L*[0] for k in range(6)]

    def feedforward(self, x):
        self.a[0] = x
        self.z[0] = x
        for k in range(1, self.L-1):
            self.z[k] = self.w[k] @ self.a[k-1] + self.b[k]
            self.a[k] = Tanh.f(self.z[k])
        self.z[-1] = self.w[-1] @ self.a[-2] + self.b[-1]
        self.a[-1] = Sigmoid.f(self.z[-1])
        return self.a[-1]

    def backprop(self, x, y):
        self.feedforward(x)
        delta = (self.a[-1] - y)
        self.dw[-1] = delta @ self.a[-2].T
        self.db[-1] = np.sum(delta, axis=1, keepdims=True)
        for k in reversed(range(1, self.L-1)):
            delta = (self.w[k+1].T @ delta) * Tanh.g(self.z[k])
            self.dw[k] = delta @ self.a[k-1].T
            self.db[k] = np.sum(delta, axis=1, keepdims=True)


#%% Inicializaciones

def xavier_init(net):
    for k in range(1, net.L):
        in_size = net.neurons[k-1]
        out_size = net.neurons[k]
        net.w[k] = np.sqrt(2/(in_size+out_size)) * rnd.randn(out_size, in_size)
        net.b[k] = np.zeros((out_size, 1))

def lecun_init(net):
    for k in range(1, net.L):
        in_size = net.neurons[k-1]
        out_size = net.neurons[k]
```

```
        net.w[k] = np.sqrt(1/in_size) * rnd.randn(out_size, in_size)
        net.b[k] = np.zeros((out_size, 1))

def he_init(net):
    for k in range(1, net.L):
        in_size = net.neurons[k-1]
        out_size = net.neurons[k]
        net.w[k] = np.sqrt(2/in_size) * rnd.randn(out_size, in_size)
        net.b[k] = 0.1*np.ones((out_size, 1))

def normal_init(net):
    for k in range(1, net.L):
        in_size = net.neurons[k-1]
        out_size = net.neurons[k]
        net.w[k] = rnd.randn(out_size, in_size)
        net.b[k] = rnd.randn(out_size, 1)

def zero_init(net):
    for k in range(1, net.L):
        in_size = neurons[k-1]
        out_size = neurons[k]
        net.w[k] = np.zeros((out_size, in_size))
        net.b[k] = np.zeros((out_size, 1))


#%% Optimizadores

def SGD(net, trng_set, val_set, loss_fn,
        epochs=30, bat_size=100, rate=1, weight_decay=0):
    x, y = trng_set
    trng_size = x.shape[1]
    for n in range(epochs):
        shf_idx = rnd.permutation(trng_size)
        x, y = x[:, shf_idx], y[:, shf_idx]
        for k in range(0, trng_size, bat_size):
            net.backprop(x[:, k:k+bat_size], y[:, k:k+bat_size])
            for k in range(1, net.L):
                decay = (1 - rate*weight_decay/trng_size)
                net.w[k] =  decay * net.w[k] - rate/bat_size * net.dw[k]
                net.b[k] =  net.b[k] - rate/bat_size * net.db[k]
        acc, loss = performance(net, val_set, loss_fn)
        print('epoch: %2d | acc: %2.2f%% | loss: %.5f' % (n+1, acc*100, loss))

def SGDMomentum(net, trng_set, val_set, loss_fn,
                epochs=30, bat_size=100, rate=0.001, momentum=0.9):
    x, y = trng_set
    trng_size = x.shape[1]
    vw, vb = [0], [0]
```

```
    for k in range(1, net.L):
        vw.append(np.zeros(net.w[k].shape))
        vb.append(np.zeros(net.b[k].shape))
    for n in range(epochs):
        shf_idx = np.random.permutation(trng_size)
        x, y = x[:, shf_idx], y[:, shf_idx]
        for k in range(0, trng_size, bat_size):
            net.backprop(x[:, k:k+bat_size], y[:, k:k+bat_size])
            for k in range(1, net.L):
                vw[k] = momentum * vw[k] - rate/bat_size * net.dw[k]
                vb[k] = momentum * vb[k] - rate/bat_size * net.db[k]
                net.w[k] =  net.w[k] + vw[k]
                net.b[k] =  net.b[k] + vb[k]
        acc, loss = performance(net, val_set, loss_fn)
        print('epoch: %2d | acc: %2.2f%% | loss: %.5f' % (n+1, acc*100, loss))


#%% Monitoreo del rendimiento

def performance(net, test_set, loss_fn):
    x, y = test_set
    test_size = x.shape[1]
    net.feedforward(x)
    pred = np.argmax(net.a[-1], axis=0)
    label = np.argmax(y, axis=0)
    acc = np.mean(pred == label)
    loss = loss_fn(net.a[-1], y) / test_size
    return acc, loss


#%% Costos

def quad_cost(a, y):
    return 1/2 * np.sum((a - y)**2)

def cross_entropy_cost(a, y):
    return -np.sum(np.nan_to_num(y*np.log(a) + (1 - y)*np.log(1 - a)))

def nll_cost(a, y):
    return -np.sum(y * np.log(a))


#%% Activaciones

class Sigmoid:
    @staticmethod
    def f(x):
        s = 1/(1 + np.exp(-x))
```

```
        return s

    @staticmethod
    def g(x):
        s = 1/(1 + np.exp(-x))
        return s * (1 - s)

class Tanh:
    @staticmethod
    def f(x):
        return np.tanh(x)

    @staticmethod
    def g(x):
        return 1 - np.tanh(x)**2

class Relu:
    @staticmethod
    def f(x):
        xx = x.copy()
        xx[xx<0] = 0
        return xx

    @staticmethod
    def g(x):
        xx = x.copy()
        xx[xx<=0] = 0
        xx[xx>0] = 1
        return xx

class Softmax:
    @staticmethod
    def f(x):
        s = np.exp(x - np.max(x, axis=0))
        return s/np.sum(s, axis=0)

    @staticmethod
    def g(x):
        s = np.exp(x - np.max(x, axis=0))
        s = s / np.sum(s, axis=0)
        I = np.eye(x.shape[0])
        return s.T * (I - s)


#%% Carga de datos

def load_data(trng_size=50000, val_size=10000,
              shuffle=False, normalize=True):
```

```
    total_size = trng_size + val_size
    x, y = np.load('../data/trng_data.npy')
    if shuffle:
        shf_idx = np.random.permutation(x.shape[1])
        x, y = x[:, shf_idx], y[:, shf_idx]
    if normalize:
        x = (x - np.mean(x, axis=0))/np.std(x, axis=0)
    trng_set = (x[:,0:trng_size], y[:,0:trng_size])
    val_set = (x[:,trng_size:total_size], y[:,trng_size:total_size])
    return trng_set, val_set

#%% Main

trng_set, val_set = load_data()
neurons = [784, 400, 100, 10]
net = Net(neurons)
xavier_init(net)
SGD(net, trng_set, val_set, cross_entropy_cost,
    epochs=30, bat_size=32, rate=0.1, weight_decay=0)
```

Bibliografía

[1] X. Glorot e Y. Bengio. «Understanding the difficulty of training deep feedforward neural networks». En: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, págs. 249-256.

[2] G. Cybenko. «Approximation capabilities of multilayer feedforward networks». En: *Mathematics of Control, Signal and Systems* 2.4 (1989), págs. 303-314.

[3] K. Hornik. «Approximation capabilities of multilayer feedforward networks». En: *Neural Networks* 4.2 (1991), págs. 251-257.

[4] J. Bergstra e Y. Bengio. «Random Search for Hyper-Parameter Optimization». En: *Journal of Machine Learning Research* 13 (2012), págs. 281-305.

[5] K. He y col. *Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification*. 2015. arXiv: 1502.01852. URL: https://arxiv.org/abs/1502.01852.

[6] I. Goodfellow, Y. Bengio y A. Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org (visitado 01-04-2019).

[7] K. P. Murphy. *Machine Learning: A Probabilistic Perspective*. The MIT Press, 2012.

[8] Stanford University. *CS231n: Convolutional Neural Networks for Visual Recognition*. 2017. URL: http://cs231n.stanford.edu (visitado 03-04-2019).

[9] J. Duchi, E. Hazan e Y. Singer. «Adaptive Subgradient Methods for Online Learning and Stochastic Optimization». En: *Journal of Machine Learning Research* 12.Jul (2011), págs. 2121-2159.

[10] G. Hinton. *Neural Networks for Machine Learning*. Coursera. 2012.

[11] D. P. Kingma y J. Ba. *Adam: A Method for Stochastic Optimization*. 2015. arXiv: 1412.6980. URL: https://arxiv.org/abs/1412.6980.

[12] Y. LeCun y col. «Gradient-based learning applied to document recognition». En: *Proceedings of the IEEE* 86.11 (1998), págs. 2278-2324.

[13] M. A. Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015. URL: http://neuralnetworksanddeeplearning.com (visitado 03-04-2019).