Borrador 2: Introducción a Redes Neuronales

Organización de Datos
Segundo Cuatrimestre 2019

9 Regresión lineal y logística

9.1 Repaso de modelos

Una manera de definir un modelo predictivo es especificar los siguientes tres componentes:

- Una hipótesis o función predictora, que es una familia de funciones
- Una métrica de error o función de costo
- Un método de optimización de los parámetros de la familia sobre la función de costo

Aclaramos que es una *hipótesis* porque no necesariamente un modelo debe funcionar bien para nuestro conjunto de datos. Por ejemplo, si sabemos que nuestros datos están relacionados a la variable objetivo por una función cuadrática, ningún modelo de regresión lineal ajustará realmente bien. Es entonces una hipótesis científica plantear un modelo por sobre el resto para ajustarlo a los datos [1].

El componente de la métrica de error nos permite comparar dos conjuntos de parámetros o incluso de hiperparámetros. Implícitamente nos dan un objetivo de optimización, que es minimizar este funcional. El último componente es esencial para el algoritmo y nos permite entrenarlo.

Este capítulo está más orientado al desarrollo de diferentes hipótesis basadas en redes neuronales. Intentaremos motivar las ideas principales y los casos nuevos de manera que las modificaciones a las hipótesis, si bien a veces son arbitrarias, tengan la mayor intuición posible.

Descenso por gradiente
El descenso por gradiente es una técnica de optimización numérica donde intentamos minimizar una función diferenciable. Recordando que el gradiente de una función diferenciable f es un vector que apunta a la dirección de máximo crecimiento, podemos maximizar el valor de la función simplemente siguiendo lo que dice el gradiente ($\bar{\nabla} f$) en cada punto. Por otro lado, si queremos minimizar el valor de la función, basta seguir el sentido opuesto ($-\bar{\nabla} f$).

Si partimos de un punto inicial $x^{(0)}$, la siguiente sucesión irá minimizando el valor que toma f en cada paso:

[1] El relector atento observará que hay familias de hipótesis que pueden aproximar cualquier función – las redes neuronales son un ejemplo. Sin embargo, esos teoremas hablan sobre la expresividad de la familia, pero no necesariamente esos parámetros se pueden *aprender* (idea de *learnability* de Valiant). Sobre todo si nuestros datos son finitos, puede no ser posible estimarlos, o encontrar mucho *overfitting*.

$$x^{(n+1)} = x^{(n)} - \gamma(n)\bar{\nabla} f(x^{(n)}) \tag{1}$$

Donde $\gamma(n)$ es el ritmo de aprendizaje, que puede ser constante o variable a lo largo de las iteraciones. Esta sucesión converge al mínimo global si la función es convexa.

9.2 El caso más sencillo

Como comienzo utilizaremos una regresión lineal para ejemplificar las tres partes de un modelo predictivo. El objetivo de este modelo es predecir en base a ciertos atributos un valor continuo, y lo hace a través de operaciones lineales. En otras palabras, sólo podemos multiplicar cada atributo por un parámetro – no es válido multiplicar atributos, elevarlos al cuadrado, etcétera. La función predictora, o hipótesis, es entonces:

$$h_\theta(x) = \theta^T x = \sum_{i=1}^{n} \theta_i x_i$$

que es una generalización n-dimensional al caso común de $y = mx + b$. Una observación a la ecuación anterior es que no hay un término independiente. Es común en las transformaciones afines **extender** los vectores para que tengan una dimensión más. En este caso, si agregamos un uno como última dimensión de x, tenemos:

$$\theta^T x = \sum_{i=1}^{n+1} \theta_i x_i + \theta_{n+1}$$

que contempla el término independiente. A lo largo de las siguientes secciones asumiremos que estamos extendiendo los vectores y las matrices.

Una métrica de error sencilla para un problema de regresión es utilizar el error cuadrático:

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{N} (h_\theta(x^{(i)}) - y^{(i)})^2$$

es importante la aclaración de que la suma es sobre los **datos**. En otras palabras, el costo definido para un conjunto de parámetros es, salvo constante, la suma del cuadrado de las diferencias entre los valores reales y los valores predichos (recordemos que estamos ante un caso de aprendizaje supervisado). La razón por la cual usamos $\frac{1}{2}$ quedará clara a continuación.

Como optimización utilizaremos el descenso de gradiente, que define la siguiente sucesión:

$$\theta_{n+1} = \theta_n - \bar{\nabla} J(\theta_n)$$

que, al reemplazar por $J(\theta)$, nos queda simplemente:

$$\theta_{n+1} = \theta_n - \sum_{i=1}^{N} x^{(i)}(h_\theta(x^{(i)}) - y^{(i)})$$

y ahora vemos por qué el $\frac{1}{2}$: para cancelar el coeficiente de la derivada del polinomio. Además, observamos que la ecuación resultante es extremadamente sencilla. Para lo que queda del resto del capítulo no elaboraremos tan detalladamente sobre las reglas de optimización (principalmente porque los métodos de clasificación tienen métricas que requieren de Probabilidad y Estadística), pero no queríamos dejar de lado un ejemplo que fije la idea. Observamos que en el estado del arte se siguen usando métodos de optimización que son variaciones del descenso de gradiente.

9.3 Regresión logística

Ahora, dados ciertos datos queremos predecir a qué clase pertenecen, en vez de estimar un valor continuo no acotado. Queremos tomar una decisión binaria. Pensemos que la regresión lineal ya nos daba una función que predecía un valor en base a ciertos atributos. Lo que queremos ahora es que el valor simplemente quede acotado entre 0 y 1. Así, valores cercanos a cero son de una clase y valos cercanos a 1 son de la otra. Entonces queremos aplicar una función $f : \mathbb{R} \to [0, 1]$. Por ejemplo:

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

esta función se llama función sigmoidea, por la forma de su curva, o función logística, por su uso. Nuestra hipótesis queda así:

$$h_\theta(x) = \frac{1}{1 + e^{-\theta^t x}} = \text{sigmoide}\left(\sum_{i}^{n+1} \theta_i x_i\right)$$

Y como métrica de error usamos el *log-likelihood* (que no describiremos) y como método de optimización seguiremos usando el descenso de gradiente.

Este modelo lo podemos representar a través de un grafo como el de la Figura 1.

9.4 Regresión softmax

La regresión logística sirve para *clasificar* si un punto pertenece a una de dos clases (o una clase y su negación). Sin embargo, es común el caso de querer predecir una de *varias* clases. Por ejemplo,

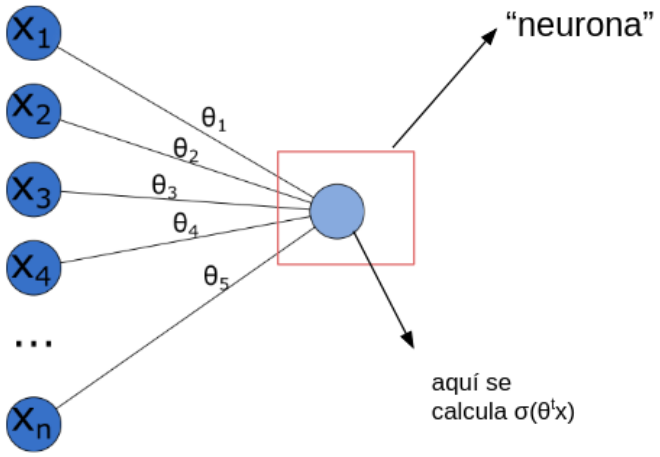

Figure 1: La ecuación de la regresión logística se puede esquematizar con este grafo. El nodo de la derecha, que aplica la función de activación al producto entre el parámetro y su entrada, le llamamos neurona.

si tenemos un sitio de venta de autos, podemos querer predecir a qué marca o modelo pertenece el auto cuya foto está en la publicación. Este, como el anterior, es un caso de predicción *exclusiva*: cada punto o dato siempre pertenece a una única clase.

¿Cómo hacemos para transformar los modelos anteriores a mecanismos de predicción que sean capaces de manejar múltiples clases? Una forma es la que aparece en el apunte en la parte de perceptrón (§12.5.7), armando un *ensamble* de predicción a partir de predictores binarios. En los párrafos siguientes vamos a generalizar la regresión logística en una regresión que predice una de varias clases.

En principio, analicemos el esquema de predicción. Hasta ahora la regresión logística emitía un escalar, que era la probabilidad de que el dato sea de clase 1. La probabilidad que sea de clase 0 era, naturalmente, la probabilidad complementaria $(1 - p)$, por ser una situación binaria. Si ahora tenemos varias clases, necesitamos un **vector de probabilidades**. Cada componente marca la probabilidad de que x sea de esa clase – por ejemplo, teniendo una función predictora de 3 clases:

$$h(x) = \begin{bmatrix} P(x \text{ sea de clase } 1) \\ P(x \text{ sea de clase } 2) \\ P(x \text{ sea de clase } 3) \end{bmatrix}$$

con la restricción adicional de que las tres componentes deben sumar 1. La regresión softmax resuelve esto definiendo un vector de parámetros para cada clase, y normalizando el vector resultante con la *función* softmax[2]:

[2]Hacemos hincapié entre la diferencia de *regresión* softmax y *función* softmax. Es la misma diferencia que está entre la regresión logística y la función logística. La última es un componente que se usa en la primera, pero aprovechamos

$$\begin{bmatrix} P(x \text{ sea de clase } 1) \\ P(x \text{ sea de clase } 2) \\ P(x \text{ sea de clase } 3) \end{bmatrix} = \text{Softmax}\left(\begin{bmatrix} \theta_1^T x \\ \theta_2^T x \\ \theta_3^T x \end{bmatrix} \right) = \frac{1}{\sum_{i=1}^{3} \exp(\theta_i^T x)} \begin{bmatrix} \exp(\theta_1^T x) \\ \exp(\theta_2^T x) \\ \exp(\theta_3^T x) \end{bmatrix}$$

¡Opa! Vayamos por partes. En principio, vemos que en cada componente tomamos la salida de $\theta_i^T x$, que es nuestra familiar regresión lineal. Luego pasamos al vector por la función softmax, que normaliza las componentes para que puedan sumar 1 (esto lo vemos en el denominador del último miembro).

¿Para qué sirven las exponenciales? La función softmax recibe su nombre por ser una forma continua y diferenciable de la operación de tomar el máximo (*soft maximum function*). En efecto, la idea es que si tenemos tres regresiones logísticas, es como tomar la predicción que arroja probabilidad más alta. Veamos esto numéricamente: supongamos que tenemos tres predictores, donde cada uno da un valor alto si el dato es de su clase o bajo en caso contrario. Digamos que las tres salidas son $[3, 4, 5]$. Ahora lo normalizamos con la función softmax, y nos da $[0.09, 0.24, 0.67]$. Softmax está asignando muchísimo más peso a la función que predijo el valor más alto, aunque la diferencia entre un valor y otro no sea tan alta proporcionalmente[3]. Esto nos importa porque a la hora de optimización vamos a multiplicar por estos valores y va a hacer que tenga más importancia el predictor 3, que es el que predijo más alto (se haya equivocado o no). En definitiva es muy similar a tomar los tres predictores y quedarnos con el que dio el valor más alto, pero la clave en este tipo de modelos es utilizar pasos diferenciables para poder optimizarlos.

Una observación más. La regresión softmax está *sobre-parametrizada*: en el caso anterior de tres clase, vimos que define los parámetros θ_1, θ_2 y θ_3. Pero la probabilidad que sea de la tercera clase es simplemente la probabilidad de que **no** sea de las clases anteriores: ¡no es necesario definir otro parámetro! Esto es aún más evidente si consideramos el caso de dos clases donde se definirían dos vectores de parámetros, aunque sabemos que se puede hacer con solo uno (como la regresión logística). Efectivamente, si incorporamos esto a la ecuación, podemos ver que softmax en dos clases se convierte en una regresión logística. Esto nos indica que la regresión softmax es una generalización de la regresión logística.

Por último, contemos la cantidad de parámetros aprendidos. Si tenemos un vector de m elementos o atributos, y k clases, la regresión softmax requiere optimizar $k \cdot m$ valores ($k \cdot (m+1)$ si incluímos el sesgo).

Podemos extender el diagrama de grafo para este nuevo modelo (Figura 2).

9.5 Perceptrón multicapa

La observación que hacemos en esta sección es que los modelos anteriores se basaban en normalizar o acotar los resultados de las regresiones lineales. Por lo tanto, esos modelos siguen siendo bastante lineales; puntualmente, sus bordes de decisión son rectas o hiperplanos. Un borde de decisión es el

para definir las funciones porque las usaremos en otras situaciones distintas de los modelos homónimos.

[3] Es fácil demostrar que es invariante a cambios por sesgo, i.e. Softmax([3,4,5])=Softmax([1003,1004,1005]). En el segundo caso la diferencia entre una componente y otra es mucho menor relativamente hablando, pero softmax sigue asignando mucho más peso a la componente máxima.

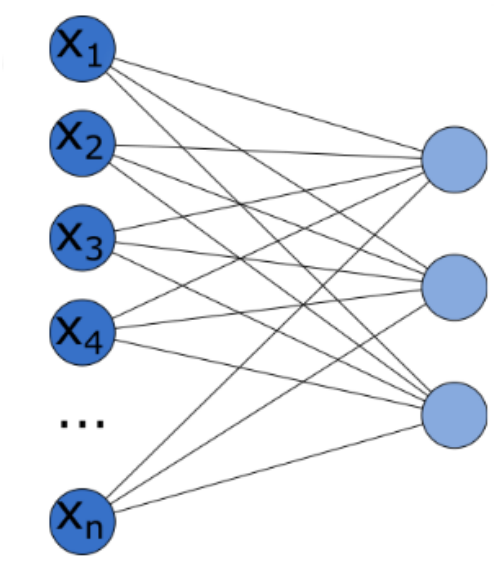

Figure 2: En este caso vemos que cada neurona tiene sus entradas y sus parámetros. Adicionalmente, normalizamos la salida de las tres neuronas con la función Softmax.

límite a partir del cual un x en el espacio de atributos pasa a ser de otra clase. Si los datos de nuestro problema no se comportan de manera lineal, los bordes de decisión pueden no ser hiperplanos y los modelos anteriores no funcionarán bien. Buscaremos formas de adaptar los modelos a ese tipo de situaciones (Figura 3).

La segunda observación fundamental para lo que sigue es entender que si un problema no separable linealmente puede transformarse por medio de una función a otro espacio donde *sí* es separable linealmente. Por ejemplo, los datos que aparecen en la Figura 3 pueden ser vistos como una proyección a dos dimensiones de un paraboloide elíptico. En ese caso, en las tres dimensiones definitivamente podemos trazar un plano que separan las clases, puesto que los puntos rojos quedarán en los z bajos y los azules en los z altos. Si para todos los casos pudiéramos hallar una transformación así, tendríamos la vida bastante resuelta: a los datos les aplicamos esta función y luego aplicamos una regresión logística o softmax según el caso. Eso podría separar cualquier set de datos.

¿Qué necesitaríamos para lograr esto? Necesitaríamos un paso intermedio de transformaciones parametrizables, de manera que los parámetros fuesen óptimos para el conjunto de datos sobre el que se opera. Junto con la parte final de regresión softmax, se pueden minimizar todos los parámetros al mismo tiempo con descenso de gradiente; no es necesario primero encontrar la mejor transformación. Sucede que una capa intermedia de neuronas — nodos de la forma *sigmoide* $\left(\sum_i^{n+1} \theta_i x_i\right)$ — cumple perfectamente estos requerimientos de transformación parametrizable. El esquema de predicción quedaría como lo muestra la Figura 4.

Es importante volver a aclarar que la capa intermedia no tiene normalización, son simplemente neuronas con sus vectores de pesos y función de normalización. Una forma de escribir matemáticamente la Figura 4 es:

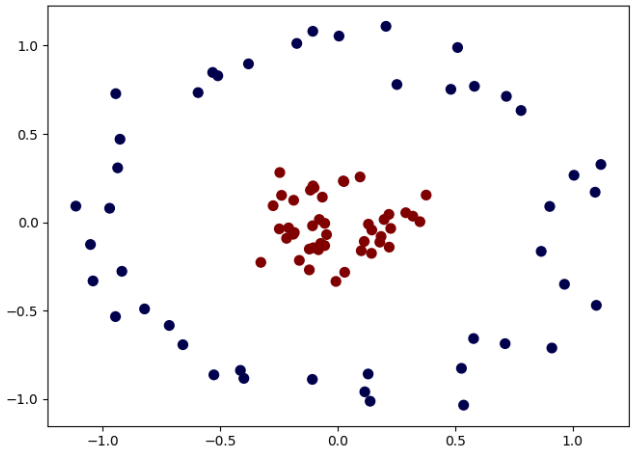

Figure 3: En este caso tenemos puntos en dos dimensiones con un límite de separación de clases no lineal (es una circunferencia, aproximadamente). Un modelo como la regresión lineal, que traza una recta para separar las clases, tendrá una precisión bastante baja en este problema.

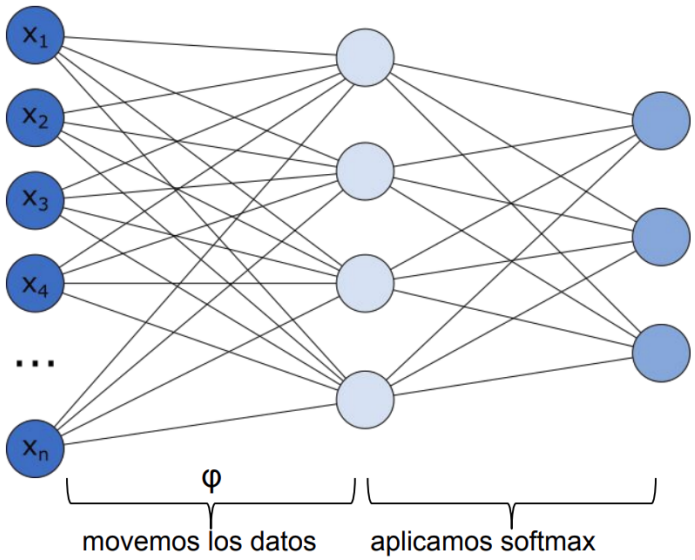

Figure 4: El esquema de predicción es agregar una capa intermedia (llamada *oculta*) de neuronas (sin normalización, sólo función sigmoidea), y luego agregar una capa de regresión softmax para predecir al final.

$$h(x) = \text{Softmax}(W_2\bar{\sigma}(W_1 x))$$

donde usamos notación matricial para abreviar los θ_i y vectorizamos la función de activación. Es decir que, para la capa intermedia de la Figura 4

$$\bar{\sigma}(Wx) = \bar{\sigma}\left(\begin{bmatrix}\theta_1^T x\\ \theta_2^T x\\ \theta_3^T x\\ \theta_4^T x\end{bmatrix}\right) = \begin{bmatrix}\sigma(\theta_1^T x)\\ \sigma(\theta_2^T x)\\ \sigma(\theta_3^T x)\\ \sigma(\theta_4^T x)\end{bmatrix}$$

Las capas de este tipo se llaman **capas densas**. Las redes neuronales que están formadas por una o varias capas intermedias (más la normalización al final) se llaman **perceptrones multicapas**.

¿Cómo sabemos que esta transformación sirve para cualquier situación? Bueno, ya que estamos:

Teorema de la aproximación universal

Sea $\phi : \mathbb{R} \to \mathbb{R}$ una función no constante, acotada, y continua (función de activación). Sea $f : \mathbb{R}^m \to \mathbb{R}$ cualquier función continua (es la transformación), y sea $\epsilon > 0$. Entonces existe $N \in \mathbb{N}, v_i \in \mathbb{R}, b_i \in \mathbb{R}$ y $w_i \in \mathbb{R}^m$ para todo i tal que:

$$F(x) = \sum_{i=1}^{N} v_i \phi(w_i^T x + b_i)$$

cumpliendo $|F(x) - f(x)| < \epsilon \; \forall x$. En otras palabras, con una capa intermedia de suficientes neuronas, podremos construir una función $F(x)$ que aproxima arbitrariamente bien a $f(x)$. Puntualmente, esto nos indica que $F(x)$ tiene una gran *expresividad*, por lo que podrá también aproximar una transformación que mueva los datos de manera que sean linealmente separables.

Otro nombre para esta idea de transformar los datos a un espacio linealmente separable, aunque no usando neuronas, es *the kernel trick* y es la base de SVM (Figura 5). Para más información sobre este concepto y el Teorema de Cover, que habla sobre mover datos a un espacio de mayores dimensiones, se puede visitar la sección §12.7 del apunte.

Recapitulando, dado cualquier conjunto de datos con sus *labels*, podremos clasificarlo con un perceptrón multicapa con una sola capa intermedia. Eso, de por sí, es un gran hito. Sin embargo, esto no nos salva del *overfitting*, o de que necesitemos una cantidad exponencial de neuronas. Tampoco nos asegura la convergencia en un tiempo razonable, o que encontremos los mejores parámetros (el teorema de la aproximación universal habla sobre la existencia de un conjunto de parámetros, no sobre cuán fácil o factible es hallarlos). Desde luego también sería posible clasificarlo con varias capas intermedias, pero la garantía ya la tenemos con una sola de suficiente tamaño.

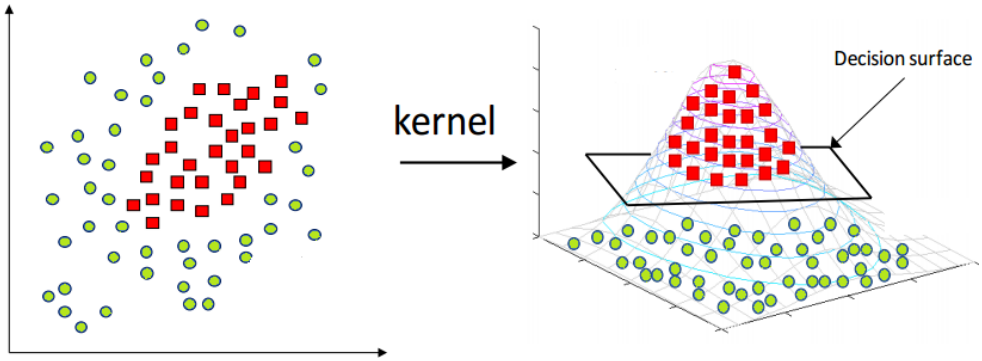

Figure 5: Un conjunto de datos que aparentemente no es separable por hiperplanos se puede mover a otro espacio donde sí es posible.

9.6 Redes convolucionales

En las secciones anteriores nos referimos a los datos siempre de manera abstracta. No habíamos especificado de qué tipo eran. Es razonable pensar que si asumimos ciertas cosas de los datos podremos modificar a los algoritmos para que funcionen mejor. En esta sección haremos justamente eso, trabajando con imágenes.

Una imagen es, en su definición más sencilla, una matriz (de tamaño 1920×1080, por ejemplo) donde cada elemento nos indica la luminosidad del píxel. Valores bajos son más oscuros y valores altos son más blancos. Si queremos representar colores, es necesario especificar una codificación – por ejemplo RGB – y usar un tensor, que se puede ver como un cubo[4]. Por ejemplo, las dimensiones de un tensor pueden ser $1920 \times 1080 \times 3$.

Si sabemos que estamos trabajando con imágenes, podemos realizar ciertas observaciones sobre su comportamiento. En principio, las imágenes tienen **localidad**. Es intuitivo pensar que píxeles cercanos pueden pertenecer a la misma cosa o ser similares. Este principio marca una diferencia fundamental con datos que son columnares: el atributo `cantidad_habitaciones` no tiene por qué ser similar al atributo `acepta_mascotas` simplemente porque quedaron juntos en la tabla.

Un perceptrón multicapa precisamente no asume tales cosas, por lo que no sirve para imágenes. Supongamos el siguiente caso. Nuestro conjunto de entrenamiento tiene imágenes donde a veces hay un auto y a veces hay no. Pero los autos solo aparecen en la región izquierda. Si nuestro conjunto de prueba tiene autos que aparecen a la derecha, el modelo definitivamente fallará en predecir correctamente tal imagen, porque un perceptrón multicapa trabaja con cada píxel como si fuese independiente del resto. Decimos entonces que este modelo **no generaliza bien**. Mejor sería utilizar un modelo que pueda capturar la noción de auto, sin importar dónde se ubique en la imagen. Después de todo, un auto es un auto, irrespectivamente de si está en la parte superior de la imagen o inferior.

Una solución a este problema es compartir parámetros. La idea es que esos parámetros se comparten

[4]Una visualización alternativa puede ser pensar al tensor como un pan lactal donde cada rebanada es una matriz. No me animé a poner esto en el cuerpo del texto.

a lo largo de la imagen; si los parámetros representan a un auto, entonces sin importar dónde esté en la imagen debería encontrarlo. Para representar esta idea de compartir parámetros vamos a introducir el concepto de **convolución de matrices**[5].

La convolución matricial toma dos matrices, generalmente una grande y una pequeña, y devuelve otra matriz. La grande será la imagen y la chica el filtro que se optimiza. Tomamos la chica y la superponemos arriba a la izquierda con la grande. Ahora multiplicamos los números que quedan superpuestos, y sumamos el resultado de los productos. Ese valor será el $(0, 0)$ de la matriz resultado. Si movemos la matriz chica una posición a la derecha y hacemos el mismo procedimiento, eso nos dará el elemento en la posición $(1, 0)$ de la matriz resultado. Si ahora bajamos la matriz chica una posición respecto de la matriz grande, nos dará la $(1, 1)$. Por ejemplo:

$$\left(\begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix} \star \begin{pmatrix} 1 & -1 \\ 0 & 2 \end{pmatrix} \right)_{1,1} = 6 \cdot 1 + 7 \cdot (-1) + 10 \cdot 0 + 11 \cdot 2 = 21$$

Que nos da el elemento (1,1) de la convolución entre las dos matrices. Efectivamente, la operación que estamos haciendo es simplemente:

$$(A \star K)_{i,j} = \sum_{y=0}^{n-1} \sum_{z=0}^{m-1} A_{i+y,j+z} \cdot K_{y,z}$$

siendo $K \in \mathbb{R}^{n \times m}$.

Démosle ahora un poco de intuición a esta operación. Pensando en imágenes, ¿qué pasaría si hacemos una convolución con el siguiente filtro?

$$\frac{1}{9} \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}$$

Lo que veríamos es que si tenemos una región de 3×3 donde el píxel central es igual a todos sus vecinos, entonces el resultado es 0. Si el píxel central es distinto al resto, entonces será distinto de cero.

¿Qué pasaría si aplicamos ese filtro a la siguiente imagen? (Figura 6)

Los píxeles de la pared que son parecidos entre sí deberían ser anulados, y los píxeles en la frente de Bill también. Sin embargo, los píxeles que están al *borde* del traje y tienen píxeles de pared arriba no se anularán. Confirmamos nuestra intuición con el resultado de la operación en la Figura 7.

[5]Si la persona que lee estas palabras es electrónica o cursó Señales y Sistemas, verá que a continuación describo una *correlación cruzada*, no una convolución. La literatura y las librerías aclaran lo mismo. De todas maneras se puede pensar que tomamos la espejada del filtro aprendido.

Figure 6: Bill Evans contento.

Figure 7: Bill Evans nos mira.

Efectivamente, ¡acabamos de armar un detector de bordes! Esto es útil porque nos independizamos de los valores que tenían los píxeles y nos quedamos con las diferencias. No importa si los anteojos eran grises, blancos u oscuros, los encontramos igual. Con esto podemos entender que las convoluciones son operaciones muy útiles en el campo de las imágenes y que permiten obtener mejores representaciones de los datos.

Fundamentalmente, la idea aquí es que el modelo optimice los parámetros de la matriz de manera que encuentre filtros que sirvan más para la tarea de predicción. Por ejemplo, en vez de detectar bordes, un filtro de un tamaño más grande (e.g. 64×64) puede detectar circunferencias y otras formas varias. Y todo esto es independiente de la ubicación en la imagen. Recordemos también que en el caso de imagenes con canales de colores, que es el caso más usual, usamos un filtro distinto para cada color. Por lo que un filtro de 64×64 se convierte en $64 \times 64 \times 3$ si usamos RGB.

Pensemos ahora en los **hiperparámetros**. Primero, el tamaño de la matriz parece ser uno. Se-

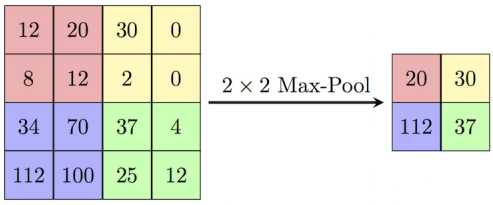

Figure 8: Operación de *max-pooling*.

gundo, ¿por qué avanzamos de a un paso? ¿Por qué no dos, o evitar directamente el solapamiento? Ese hiperparámetro se llama *stride*. Por último, un detalle importante: en los ejemplos numéricos anteriores obtuvimos una matriz más pequeña que la imagen original. Si no queremos que esto suceda, podemos agregar *padding* a la misma de manera que el resultado de la operación quede de un tamaño igual a la imagen. La decisión de tomar padding o no, y qué tipo de padding (e.g. ¿ponemos ceros? ¿Repetimos píxeles?) es otro hiperparámetro.

Otra operación ubicua en este tipo de redes es *max-pooling*. La idea es reducir el tamaño de la imagen quedándonos con los *features* más importantes. Se define un tamaño de bloque, por ejemplo 2×2, y nos quedamos con el píxel de valor más alto. También recorremos la imagen con este bloque como hacíamos con la convolución, generando la imagen de resultado. Lo curioso de este método es que si bien tiene hiperparámetros, **no** tiene parámetros. Opciones alternativas son *average-pooling* donde en vez de tomar el máximo tomamos el promedio.

Armemos una intuición sobre qué implica concatenar sucesivamente este tipo de operaciones. La convolución, como vimos, genera píxeles nuevos que se refieren a regiones de la imagen anterior. Si aplicamos una capa convolución nuevamente, encontraremos que los píxeles nuevos se refieren a regiones de la capa anterior, y en definitiva son regiones más grandes de la imagen original. De alguna manera, este operador nos permite difundir información a lo largo de la imagen. Por otro lado, la operación de *max-pooling* nos permite reducir el tamaño de la imagen para quedarnos con un "resumen" de los píxeles más importantes.

En la Figura 9 veremos una arquitectura de ejemplo para una red convolucional. Lo que se grafica es el tamaño del tensor en cada parte. Dos detalles: en una capa convolucional sería absurdo usar un solo filtro – por lo que usamos varios. A eso nos referimos con *depth*. Las capas finales son *fully connected* (FC), que son las densas que vimos en la sección anterior. Con eso podemos clasificar, pero primero tuvimos que lograr una mejor representación de los datos para poder extraer *features* relevantes. La idea es que a esas capas no les llegue el valor de los píxeles, sino atributos más interesantes como "hay un rectángulo" o "esto parece una corbata".

9.7 Redes profundas (*deep learning*)

El teorema de aproximación universal nos garantizaba que tener una red neuronal densa de una sola capa intermedia, con una cantidad grande de neuronas, era suficiente para poder clasificar cualquier conjunto de datos. Si bien esto es cierto, no siempre es la mejor opción. Pensemos que el objetivo de las capas intermedias es generar una buena representación de los datos – en definitiva, son **abstracciones**. En nuestro caso, como humanos, nos sirve mucho más armar abstracciones en

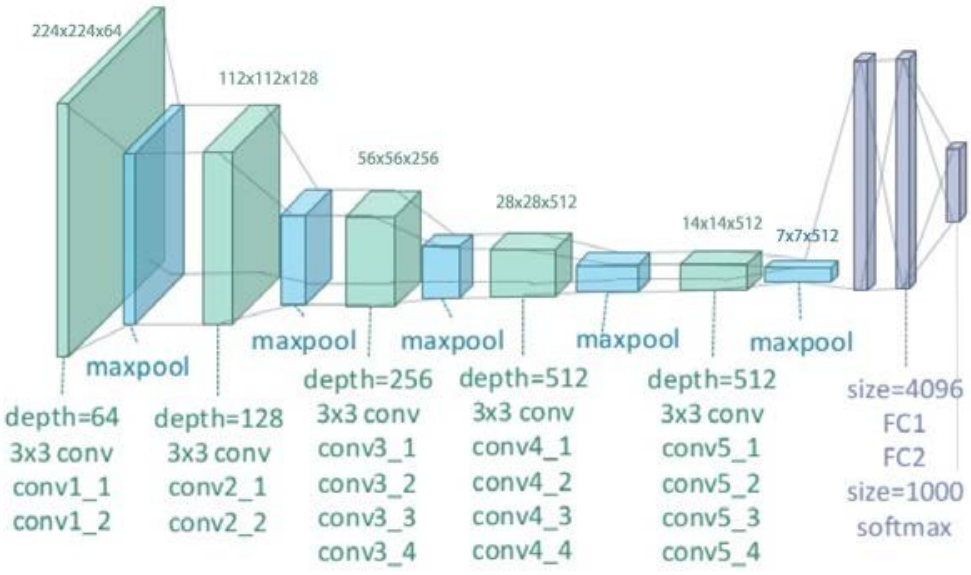

Figure 9: Ejemplo de arquitectura de una red convolucional.

cadena, en vez de hacer todo en un solo paso. Si extendemos esta idea a las redes neuronales, la conjetura es que posiblemente necesitemos muchas menos neuronas totales (y por ende parámetros) si utilizamos construcciones **profundas** en vez de **anchas**. Preferiremos entonces utilizar varias capas para descubrir *features* en vez de una sola. Es precisamente lo que vemos en la Figura 9.

En la práctica no todo es viento en popa. Sucede que al entrenar la red profunda, vemos que cada vez aprende más lento. Abriendo el *debugger* e inspeccionando los parámetros para ver cómo evolucionan, ¡vemos que varían muy poco! (Figura 10).

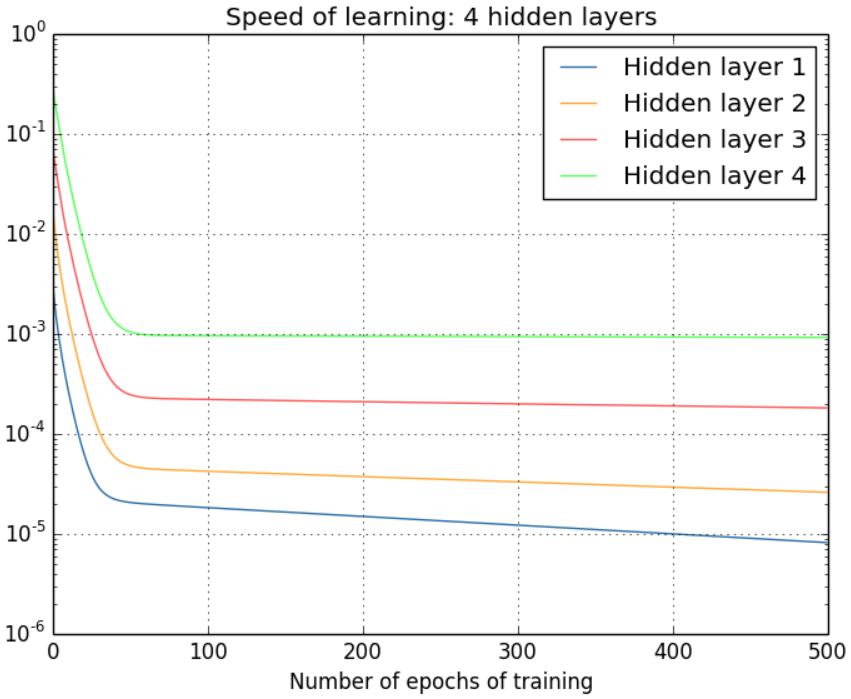

Figure 10: Las capas más alejadas de la salida varían exponencialmente menos sus parámetros..

Retropropagación y los gradientes desvanecientes

Las redes neuronales se entrenan a través de un algoritmo que se llama **retropropagación**. Sabiendo que una red neuronal de varias capas se puede escribir de la forma:

$$f(x) = \text{Softmax}(capa_n(capa_{n-1}(\cdots capa_1(x))))$$

el algoritmo de retropropagación nos da una forma de calcular los gradientes de cada capa con una regla de la cadena eficiente. Si como función de activación utilizamos las sigmoides, en algún paso estará la derivada de esa función:

$$\frac{\mathrm{d}}{\mathrm{d}x}\frac{1}{1+\exp(-x)} = \frac{e^{-x}}{(e^{-x}+1)^2}$$

que es una función que alcanza su máximo en 0 y vale $\frac{1}{4}$. Esto significa que al aplicar la regla de la cadena para hallar el gradiente de las capas anteriores, en cada paso iremos multiplicando por a lo sumo $\frac{1}{4}$. Si tenemos 6 capas, los coeficientes del gradiente de la primera capa serán como mucho $\frac{1}{4^6}$, que es un número pequeñísimo. ¡Ahora entendemos la Figura 10!

Para disminuir el problema de los gradientes desvanecientes, usamos funciones de activaciones con derivadas que se comporten mejor. Por ejemplo, *LeakyReLU*:

$$f(x) = \begin{cases} x & \text{si } x > 0 \\ 0.01x & \text{caso contrario.} \end{cases}$$

Esta función tiene derivadas que no decrecen exponencialmente en ambos lados, por lo que al aplicarlo en la regla de la cadena no estaremos siempre multiplicando por valores menores a 1.

A continuación detallamos técnicas adicionales para lograr un entrenamiento mejor de la red neuronal:

Regularización. Como vimos en otras clases, la regularización es una buena manera de evitar el *overfitting* introduciendo penalidades en el tamaño de los parámetros aprendidos. Métodos comunes son utilizando métricas L1 o L2.

Drop-out. Experimentalmente se observa que hay neuronas dentro de la red que aprenden más y son más importantes en la tarea de predicción, mientras que hay otras con pesos muy pequeños que no tienen mucho efecto. La técnica de *drop-out* hace que durante el entrenamiento no se usen todas las neuronas todo el tiempo. De esta manera, se optimizan más equitativamente todos los parámetros con la idea de que se obtenga mejores resultados de predicción. Esta idea de bloquear parte de las neuronas también se puede usar en la capa inicial de datos, que sería equivalente a bloquear algunas regiones o píxeles de la imagen para obligar a que la red pueda inferir de todas maneras la clase correcta.

Batch normalization. Esta técnica normaliza la entrada a cada capa. La idea es que, como siempre, normalizar puede ayudar a predecir mejor. Una observación es que si no se normaliza, un dato con una escala un poco más alto se va potenciando con cada capa, dificultando el trabajo de las capas finales que pueden recibir escalas muy distintas. Al normalizar en cada capa evitamos este problema.

Layer normalization. Esta técnica normaliza los *pesos* de cada capa. Fue desarrollado para mejorar los resultados y la performance de **batch normalization**.

Transfer learning. No siempre tenemos millones de imágenes para nuestro conjunto de entrenamiento, pero podemos querer aplicar una red convolucional. La idea es que podemos tomar una red convolucional ya entrenada para otro conjunto de datos y volver a entrenarla para nuestro conjunto de datos en particular. Esto implica mantener las capas convolucionales y re-armar la parte *fully connected* para que se pueda aplicar a nuestro dominio. Intuitivamente, este método está aprovechando que los filtros aprendidos por la red en otro problema pueden ser útiles en el dominio nuevo.

9.8 Referencias

Nielsen, M. A. (2015). Neural networks and deep learning (Vol. 25). San Francisco, CA, USA:: Determination press.

Olah, C. (2014). Neural Networks, Manifolds, and Topology. Colah's blog.

Olah, C. (2015). Calculus on Computational Graphs: Backpropagation. Colah's blog.

Ng, A. et al, Unsupervised Feature Learning and Deep Learning tutorial, Stanford.

Kurita, K. (2018). An Intuitive Explanation of Why Batch Normalization Really Works. Machine Learning Explained Blog.