Entrenamiento

Nicolás Kossacoff

Noviembre 2024

1. Introducción

El entrenamiento de una red neuronal consiste en encontrar los pesos que devuelven el modelo que mejor ajusta a los datos, es decir, que minimizan la función de pérdida.

El procedimiento es sencillo. Inicializamos los parámetros del modelo de manera aleatoria y luego iteramos sobre los siguientes pasos:

1. Computamos las derivadas (gradientes) de la función de pérdida con respecto a los pesos. De esta manera podemos saber como cambia la función al modificar los parámetros.
2. Ajustamos los parámetros siguiendo la dirección de los gradientes para reducir la función de pérdida.

2. Algoritmos de optimización

Los algoritmos de optimización buscan los pesos que minimizan la función de pérdida de nuestra red neuronal:

$$\hat{W} = \arg\min_{W} L(W)$$

Los pesos se inicializan de manera heurística y se van ajustando de manera que la función de pérdida se reduzca después de cada iteración.

2.1. Gradiente descendente:

Este algoritmo inicializa aleatoriamente los pesos, $W_0 = (w_1, \ldots, w_N)'$, e itera sobre los siguientes pasos:

1. Computa la derivada de la función de pérdida respecto de los pesos actuales:

$$\frac{\partial L}{\partial W} = \left(\frac{\partial L}{\partial w_1}, \ldots, \frac{\partial L}{\partial w_N} \right)$$

2. Actualizamos los pesos de acuerdo a la siguiente regla:

$$W_1 \longleftarrow W_0 - \alpha \frac{\partial L}{\partial W} \tag{1}$$

donde α es la tasa de entrenamiento y determina en que medida actualizamos los pesos (i.e., cuanto nos movemos en la dirección en la que se reduce la función de pérdida).

En el mínimo de la función de pérdida, los gradientes son iguales a cero y los parámetros dejan de modificarse (no hay ninguna dirección en la que podamos mejorar). Sin embargo, en la práctica esto no ocurre y se termina el procedimiento cuando los gradientes son muy cercanos a cero.

2.1.1. Ejemplo: regresión lineal

Supongamos que tenemos un modelo de regresión lineal, $f(x, W) = \beta_0 + \beta_1 x$, con la siguiente función de pérdida:

$$L(W) = \sum_{i=1}^{n} \left(f(x_i, W) - y_i\right)^2 = \sum_{i=1}^{n} \left(\beta_0 + \beta_1 x_i - y_i\right)^2$$

La derivada de $L(W)$ con respecto a los pesos de nuestro modelo es:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial f(\bullet)} \frac{\partial f(\bullet)}{\partial W}$$

La figura 1 nos muestra la progresión de este algoritmo cuando computamos las derivadas de la función de pérdida respecto a los pesos, como en la ecuación anterior, y ajustamos los pesos según la ecuación (1).

2.1.2. Mínimos locales

Las funciones de pérdida para los problemas de regresión lineal siempre tienen mínimos globales bien definidos. En particular, son funciones convexas, lo que nos asegura que sin importar donde inicializamos los pesos de nuestro modelo siempre vamos a alcanzar el mínimo global al movernos en la dirección contraria al gradiente.

Por otro lado, las funciones de pérdida de los modelos no lineales, incluyendo a las redes neuronales superficiales y profundas, son no convexas. Estas funciones no convexas tienen muchos **mínimos locales**, y como inicializamos nuestros pesos de manera aleatoria, nada nos garantiza que el algoritmo no converja a uno de ellos[1]. Más aún, no hay manera de saber si llegamos al mínimo global o si hay una mejor solución posible.

También nos podemos encontrar con **puntos de ensilladura (saddle-points)** en donde los gradientes son cero pero la función crece en algunas direcciones y decrece en otras. Si los pesos actuales no se encuentran justo sobre este punto, entonces es posible escapar en alguna dirección. Sin embargo, su entorno cercano es plano, por lo que es difícil comprobar que el algoritmo no converge porque nos encontramos en un saddle-point.

[1] Sin embargo, para redes neuronales grandes, la mayoría de los mínimos locales son similares y presentan rendimientos similares en las muestras de validación. Es decir, la probabilidad de encontrar mínimos locales malos decrece con el tamaño de la red.

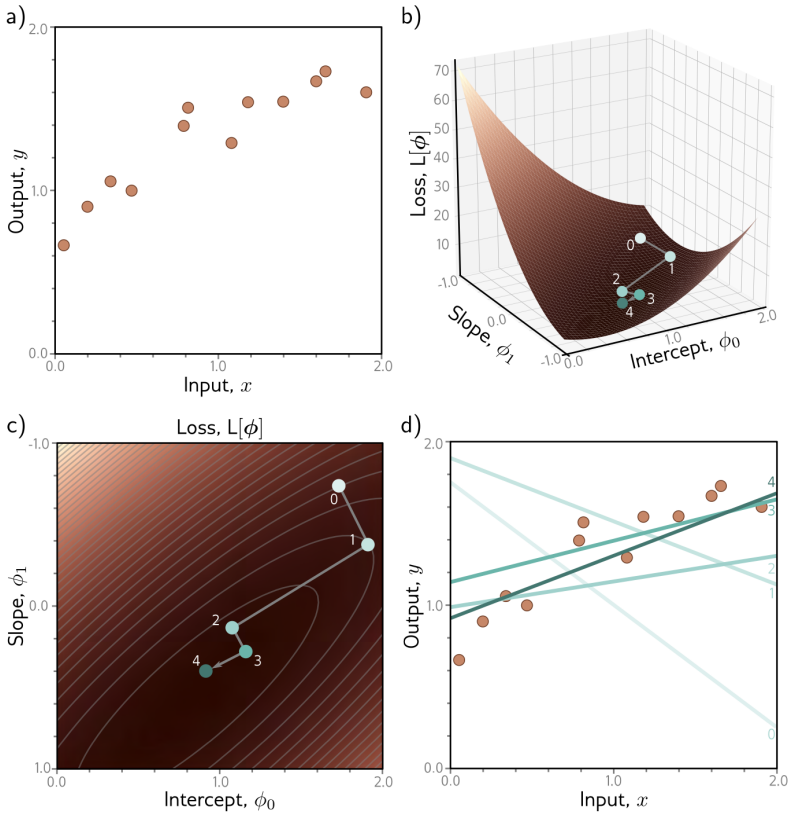

Figura 1: *Figura c)* Con los pesos iniciales la función de pérdida se posiciona sobre el punto cero. Si computamos los gradientes, actualizamos los pesos según la ecuación (1) y volvemos a calcular la función de pérdida, nos movemos hacia el punto uno. En este punto la función de pérdida es menor. Repetimos el procedimiento hasta llegar al punto cuatro, donde la función de pérdida se minimiza. *Figura d)* Cada recta nos muestra el modelo que obtenemos después de cada iteración. Se puede apreciar como el modelo final es el que mejor ajusta a los datos. **Source:** Price, S. (2024). *Understanding Deep Learning* [Figura 6.1, p. 79]

2.2. Gradiente Descendente Estocástico (SGD)

Uno de los principales problemas del algoritmo de gradiente descendente es que la solución depende completamente de nuestro punto de partida. El algoritmo de **Gradiente Descendente Estocástico** busca solucionar este problema agregando 'ruido' al gradiente en cada iteración.

Debido al ruido que agregamos a nuestra estimación del gradiente, el trayecto de nuestra solución no es determinístico (i.e., partir dos veces de un mismo punto de partida no nos devuelve el mismo resultado). En promedio, la solución se mueve en la dirección que reduce la función de pérdida pero, debido al ruido que introducimos, hay iteraciones en las que se mueve en la dirección contraria. Esto nos ayuda a evitar caer en mínimos locales o saddle-points, como se puede observar en la figura 2.

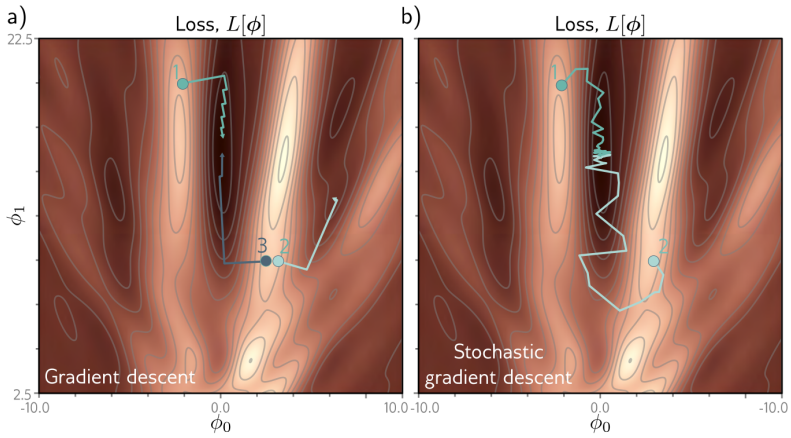

Figura 2: *Figura a)* Podemos observar como el punto de partida determina el resultado final del algoritmo. Los puntos dos y tres son puntos de partida cercanos, pero uno de ellos termina en el mínimo global (punto tres) y el otro termina en un mínimo local (punto dos). *Figura b)* Al permitir, en algunas iteraciones, que la solución se mueva en la dirección de mayor crecimiento, logramos movernos de una región a otra, evitando que la solución del punto dos termine en un mínimo local. **Source:** Price, S. (2024). *Understanding Deep Learning* [Figura 6.5, p. 83]

2.2.1. Batches & epochs

Lo que hace el algoritmo para agregar ruido es, en cada iteración, elegir aleatoriamente un sub-conjunto (llamados **mini-batch** o **batch**) de la muestra de entrenamiento y computar el gradiente solo para esas observaciones. Estos gradientes son una estimación de los verdaderos gradientes, lo que introduce ruido al proceso.

Los pesos de nuestro modelo en la iteración t se actualizan de la siguiente manera:

$$W_{t+1} \longleftarrow W_t - \alpha \sum_{i \in B_t} \frac{\partial l_i}{\partial W}$$

donde l_i es la función de pérdida para la i-ésima observación y B_t es el batch en la iteración t.

Los batch se extraen de la muestra de entrenamiento de manera aleatoria y sin reposición. El algoritmo extrae batchs hasta que haya utilizado toda la muestra de entrenamiento, y vuelve a comenzar. Una pasada por toda la muestra de entrenamiento se conoce como una **época (epoch)**.

3. Backpropagation

La derivada de la función de pérdida con respecto a los pesos nos dice como cambia la función debido a un pequeño cambio en los pesos. Los algoritmos de optimización vistos anteriormente hacen uso de esta información para ajustar los pesos, minimizar la función de pérdida y obtener el modelo que mejor ajusta a nuestros datos.

El algoritmo de backpropagation se encarga de calcular estas derivadas. Es importante tener en cuenta dos cosas:

1. Cada peso multiplica el resultado (activación) de una neurona en una capa oculta para que pueda ser utilizado como el valor de entrada de la siguiente capa oculta. Debido a esto, el efecto de un cambio en uno de los pesos se puede ver atenuado o amplificado por la función de activación que estemos aplicando en cada neurona.

   Previo a calcular los gradientes, corremos la red neuronal para cada observación en el batch y guardamos las activaciones de cada neurona. Esto se conoce como la pasada **forward**.

2. Un pequeño cambio en uno de los pesos tiene efecto sobre el resto de la red neuronal. Por ejemplo, si cambiamos un de los pesos que alimenta a la primer capa oculta (h_1), tenemos que entender: (i) como ese cambio en h_1 afecta a la segunda capa (h_2), (ii) como el cambio en h_2 afecta a la tercer capa (h_3), y así hasta que llegamos al final de la red.

   Por lo tanto, para saber como cambia la función de pérdida al cambiar uno de los pesos, también necesitamos saber como este cambio afectará al resto de la red neuronal. Lo bueno es que, como esta información también es necesaria al considerar otros pesos en la misma capa o en capas previas, pueden ser reutilizadas, lo que lo hace muy eficiente.