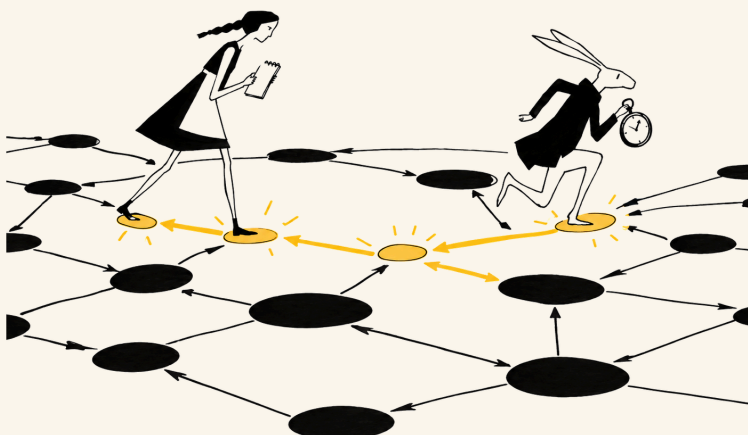


Las aventuras de Alicia en el País de las Maravillas Diferenciables



Simone Scardapane · 2024

Audiolecciones ilustradas

Lecturized

Las aventuras de Alicia en el País de las Maravillas Diferenciables

Audiolecciones ilustradas

Simone Scardapane · 2024

Contado como lecciones ilustradas por Lecturized

lecturized.com

Del original: Alice's Adventures in a Differentiable Wonderland, Volume I: A Tour of the
Land

Índice

Un programa hecho de pasos pequeños

Aplanar las tablas y enunciar la tarea

Tres formas de obtener una derivada

Darle la vuelta a la cadena

Ahorrar memoria y pedir solo productos

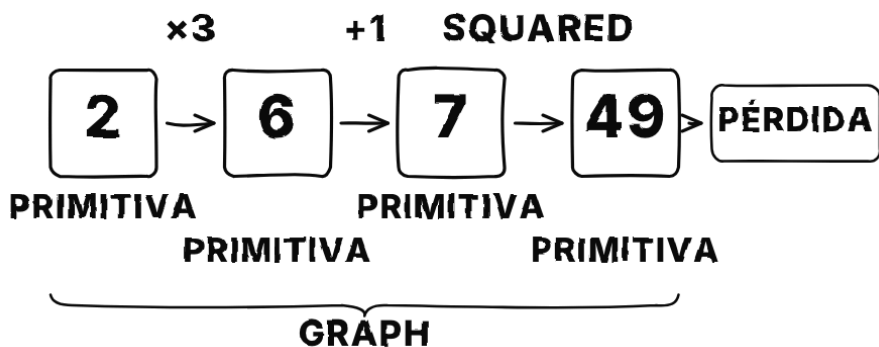
El paso hacia atrás de una capa

Los marcos de trabajo, y por qué gana la ReLU

La esquina de la ReLU

Un programa hecho de pasos pequeños

Abra una hoja de cálculo y construya una pequeña cadena de celdas. En la primera celda escriba un número, por ejemplo 2; en la segunda, una fórmula que triplique la primera; en la tercera, una que sume 1 a la segunda; en la última, una que eleve al cuadrado la tercera. La última celda muestra 49, porque 2 por 3 es 6, más 1 es 7, y 7 al cuadrado es 49. Ahora hágase una pregunta más fina: si sube un poco la primera celda, ¿con qué rapidez se mueve la última? Podría probarlo a mano, escribiendo 2,01 y mirando el resultado, pero la hoja de cálculo ya conoce cada paso que dio, y cada paso por separado es fácil de derivar.



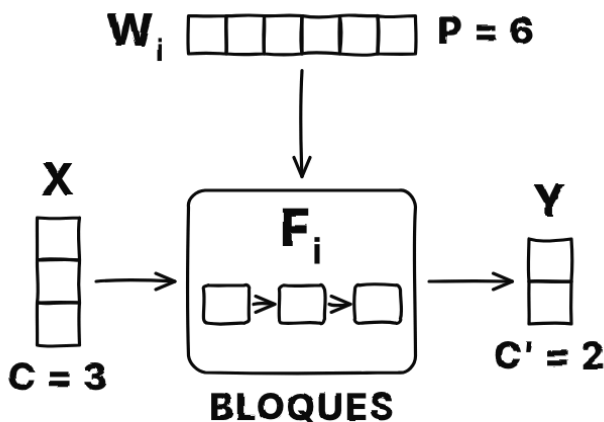
Una cadena de pasos pequeños

Ese es exactamente el problema que aborda este capítulo del libro. Cuando se entrena un asistente de chat, un enorme programa de pasos así convierte una frase en un único número, la pérdida, que dice cuánto se equivocó la respuesta. Para mejorar el modelo, el ordenador necesita el gradiente: cómo cambia la pérdida cuando se mueve cada uno de millones de números ajustables. El libro llama a la tarea de obtener estos gradientes de forma eficiente, para cualquier programa, diferenciación automática, abreviada AD. El método hacia el que avanza tiene dos nombres: retropropagación entre quienes trabajan con redes neuronales, y diferenciación automática en modo inverso entre los informáticos.

El autor nos pide imaginar un grafo computacional: el conjunto de operaciones pequeñas y atómicas que se obtiene al ejecutar de verdad el programa, como las celdas de nuestra hoja de cálculo. Cada una de estas operaciones se llama primitiva. Por brevedad, el libro sigue una sola cadena de primitivas, una tras otra, y señala que todo se traslada a grafos que se ramifican, siempre que no tengan bucles, e incluso a otros más generales.

Al principio el problema parece ya resuelto. La regla de la cadena de los jacobianos, vista antes en el libro, dice que la derivada de una composición de funciones es simplemente el producto de sus matrices jacobianas, una por paso. Un jacobiano es una tabla de derivadas parciales: una fila por cada número de salida, una columna por cada número de entrada, y cada casilla dice cuánto se mueve esa salida cuando se empuja esa entrada. Entonces, ¿por qué un capítulo entero? Porque hacer este producto con inteligencia, sin malgastar tiempo ni memoria, es el verdadero reto, y el algoritmo que lo resuelve es una piedra angular de las redes neuronales y de la programación diferenciable en general. El autor añade que entenderlo explica también cómo están diseñados entornos como TensorFlow, PyTorch y JAX, y por qué se diferencian.

Así prepara el libro sus piezas. Cada primitiva se escribe como $\mathbf{y} = f_i(\mathbf{x}, \mathbf{w}_i)$. En palabras: la i -ésima primitiva toma un vector de entrada $\mathbf{x}$ y un vector de números ajustables $\mathbf{w}_i$, sus pesos, y produce un vector de salida $\mathbf{y}$. Las formas se escriben con una tilde: $\mathbf{x} \sim (c_i)$, $\mathbf{w}_i \sim (p_i)$ y $\mathbf{y} \sim (c'_i)$. ¿De dónde sale esta forma? Piense en una proyección lineal dentro de una capa de una red: la entrada viene del paso anterior, los pesos pertenecen al propio paso y el resultado se pasa al siguiente. Con números: una capa pequeña que recibe 3 números y entrega 2, y que tiene 6 pesos, tiene c igual a 3, c' igual a 2 y p igual a 6.

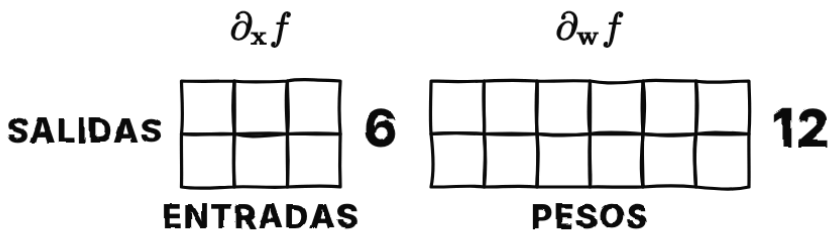


Una primitiva: entrada, pesos, salida

La definición es deliberadamente amplia. Una primitiva puede ser una sola multiplicación de matrices, una capa entera como las que construyó el capítulo anterior, por ejemplo una capa totalmente conectada con su función de activación, o un bloque o modelo completo. Bloques dentro de bloques: esta componibilidad recursiva es una propiedad clave de la programación, y el libro la conserva.

Lo único que se supone de cada primitiva es que sabemos calcular sus derivadas parciales respecto de sus dos argumentos. La primera es el jacobiano de entrada, $\partial_x[f(x, w)] \sim (c', c)$: cómo se mueve cada salida cuando se empuja cada entrada. La segunda es el jacobiano de los pesos, $\partial_w[f(x, w)] \sim (c', p)$: cómo se mueve cada salida cuando se empuja cada peso. Las formas salen de la regla de una fila por salida y

una columna por cada cosa que se empuja. Para nuestra capa pequeña, el jacobiano de entrada es una tabla de 2 filas y 3 columnas, 6 números, y el jacobiano de los pesos tiene 2 filas y 6 columnas, 12 números.



Los dos jacobianos de una capa

Estas suposiciones son razonables, dice el autor, porque el libro se limita a modelos diferenciables. Una primitiva continua pero con algunos puntos sin derivada, como la ReLU, que conserva los números positivos y pone a cero los negativos, se encaja más adelante con una herramienta llamada subgradiente. Las operaciones que no son diferenciables en absoluto, como sacar una muestra aleatoria o cortar en un umbral, también pueden tratarse relajando su gradiente

o usando un estimador equivalente; el autor lo deja para el próximo volumen.

Para recordar

Entrenar una red requiere el gradiente de un número, la pérdida, respecto de cada peso, y la diferenciación automática es la tarea de calcularlos todos de forma eficiente para cualquier programa. Un programa, una vez ejecutado, es una cadena de primitivas pequeñas, cada una escrita $y = f_i(\mathbf{x}, \mathbf{w}_i)$: una entrada, sus propios pesos, una salida. Para cada primitiva solo necesitamos dos tablas de derivadas parciales, el jacobiano de entrada, de forma c' por c , y el jacobiano de los pesos, de forma c' por p . La regla de la cadena ya dice que la respuesta es un producto de jacobianos; toda la dificultad, y la razón por la que importa la retropropagación, está en calcular ese producto de forma barata.

Preguntas

1. La regla de la cadena ya dice que la derivada de una composición es un producto de jacobianos. ¿Por qué necesita el libro todo un capítulo sobre diferenciación automática?
 - a. Porque la regla de la cadena no vale para las redes neuronales
 - b. Porque calcular ese producto con inteligencia, sin malgastar tiempo ni memoria, es el verdadero reto

- c. Porque los jacobianos de las primitivas suelen ser desconocidos
 - d. Porque la pérdida es un vector y la regla de la cadena necesita un escalar
2. Una primitiva recibe una entrada de c números, tiene p pesos y da una salida de c' números. ¿Qué forma tiene su jacobiano de los pesos?
- a. c filas y p columnas
 - b. c' filas y c columnas
 - c. c' filas y p columnas
 - d. p filas y c' columnas
3. ¿Qué supone el libro que sabemos de cada primitiva?
- a. Cómo calcular sus derivadas parciales respecto de su entrada y de sus pesos
 - b. Solo cómo calcular su salida, nada sobre derivadas
 - c. Su fórmula simbólica completa, para que un sistema de álgebra computacional trabaje con ella
 - d. Que es una sola multiplicación de matrices
4. ¿Cómo encaja el libro en este marco una primitiva como la ReLU, que tiene un punto sin derivada?
- a. Descarta la ReLU y usa solo funciones suaves
 - b. Deja la ReLU para el próximo volumen junto con el muestreo
 - c. Usa una herramienta llamada subgradiente para los puntos sin derivada
 - d. Sustituye la ReLU por una aproximación numérica de su derivada

Respuestas

1. **b)** La respuesta es, en principio, un producto de jacobianos; la dificultad, y la razón por la que importa la retropropagación, está en hacer ese producto de forma eficiente.
2. **c)** Un jacobiano tiene una fila por salida y una columna por cada cosa que se empuja; en el jacobiano de los pesos lo que se empuja son los p pesos, así que su forma es c' por p.
3. **a)** La única suposición es que cada primitiva viene con su jacobiano de entrada y su jacobiano de los pesos.
4. **c)** Las primitivas continuas con algunos puntos no diferenciables se encajan con subgradientes; solo las operaciones totalmente no diferenciables, como el muestreo, quedan para el próximo volumen.

Ejemplos resueltos

El tamaño de los dos jacobianos de una capa Una primitiva $y = f(x, w)$ recibe una entrada de 3 números, tiene 6 pesos y devuelve 2 números. ¿Cuántas casillas tienen su jacobiano de entrada y su jacobiano de los pesos?

Lea los tamaños: $c = 3$ entradas, $p = 6$ pesos, $c' = 2$ salidas.

El jacobiano de entrada tiene una fila por salida y una columna por entrada, así que su forma es $(c', c) = (2, 3)$.

Eso da $2 \times 3 = 6$ casillas, cada una dice cuánto se mueve una salida cuando se empuja una entrada.

El jacobiano de los pesos tiene una fila por salida y una columna por peso, así que su forma es $(c', p) = (2, 6)$, es decir, $2 \times 6 = 12$ casillas.

Respuesta: El jacobiano de entrada tiene 6 casillas y el de los pesos, 12.

Ejercicios

1. Una primitiva recibe 4 números, tiene 12 pesos y entrega 3 números. Escriba las formas de su jacobiano de entrada y de su jacobiano de los pesos.

Pista: Una fila por salida; una columna por entrada en el primero, por peso en el segundo.

Respuesta: Jacobiano de entrada (3, 4), jacobiano de los pesos (3, 12): las salidas dan las filas; las entradas o los pesos, las columnas.

2. Explique con sus propias palabras por qué al libro le parece bien llamar «primitivas» tanto a una sola multiplicación de matrices como a una capa entera o a un modelo completo.

Pista: Piense en qué necesita saber realmente el método de cada paso.

Respuesta: El método solo necesita, para cada paso, su salida y sus dos jacobianos, así que da igual lo grande que sea el paso. Un bloque hecho de bloques más pequeños sigue siendo un paso con una entrada, unos pesos y una salida; esta componibilidad recursiva, una propiedad clave de la programación, permite aplicar el mismo razonamiento a cualquier escala.

3. Entrenar un asistente de chat convierte una frase en un único número, la pérdida. ¿Qué tiene que entregar la diferenciación automática para esa pérdida, y por qué es eso lo que necesita el entrenamiento?

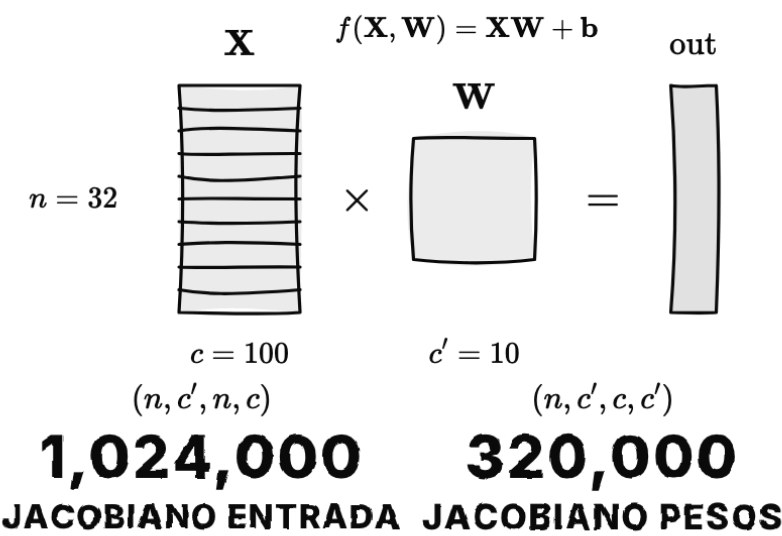
Pista: ¿Qué debe saber el ordenador de cada número ajustable antes de poder mejorar el modelo?

Respuesta: Debe entregar el gradiente de la pérdida respecto de cada peso, para todos los pesos a la vez y de forma eficiente. El entrenamiento cambia cada peso en la dirección que baja la pérdida, y el gradiente es justamente lo que dice cómo se mueve la pérdida cuando se empuja cada peso.

Aplanar las tablas y enunciar la tarea

Imagine una hoja de cálculo con 32 filas y 100 columnas: cada fila es una foto de su teléfono, cada columna un número que la aplicación de fotos ha medido sobre ella. La aplicación no mira las fotos de una en una; toma la hoja entera, un minilote, de una vez. Ahora pregúntese cuántas derivadas parciales describen una sola capa que trabaja sobre esa hoja. Quizá espere unos cientos. La cuenta honrada llega a cerca de un millón, y ver por qué es la primera lección de esta parte.

UN LOTE POR UNA CAPA

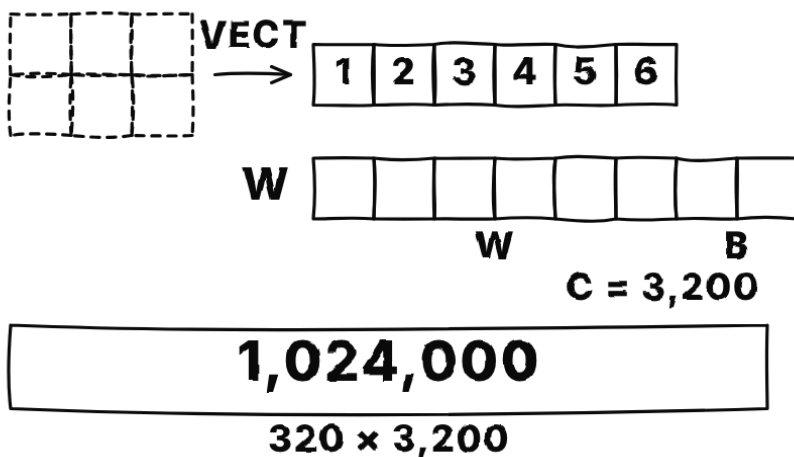


Un lote a través de una capa

Hasta ahora el libro ha hablado de vectores, de modo que cada jacobiano es una tabla plana, una matriz. Las primitivas reales a menudo reciben y entregan arreglos de rango mayor, es decir, con más de dos direcciones. El ejemplo del libro es una capa totalmente conectada sobre un minilote: $f(\mathbf{X}, \mathbf{W}) = \mathbf{XW} + \mathbf{b}$. En palabras: el lote $\mathbf{X}$ se multiplica por la matriz de pesos $\mathbf{W}$, y el vector de sesgo $\mathbf{b}$ se suma a cada fila. La entrada $\mathbf{X}$ tiene forma (n, c) , con n ejemplos de c números cada uno; $\mathbf{W}$ tiene forma (c, c') y $\mathbf{b}$ tiene forma (c') , donde c' , el número de salidas, es un hiperparámetro, un tamaño que elegimos; la salida tiene forma (n, c') .

¿De dónde salen las formas de los jacobianos? Un jacobiano empareja cada número de salida con cada número que se empuja. La salida tiene dos direcciones, n y c' ; la entrada tiene dos, n y c ; así que el jacobiano de entrada tiene forma (n, c', n, c) , un arreglo de rango 4. El jacobiano de los pesos, que empareja la salida con la matriz $\mathbf{W}$, tiene forma (n, c', c, c') , también de rango 4. Con la hoja de fotos, n es 32 y c es 100; tome c' igual a 10. El jacobiano de entrada contiene entonces 32 por 10 por 32 por 100 casillas, que son 1 024 000; el jacobiano de los pesos contiene 32 por 10 por 100 por 10, que son 320 000.

El libro escapa de los rangos altos con un truco sencillo: aplanar. Lea el arreglo fila tras fila y coloque todos sus números en un solo vector largo, que se escribe $\mathbf{x} = \text{vect}(\mathbf{X})$. Los pesos también se convierten en un vector, la matriz aplanada seguida del sesgo: $\mathbf{w} = [\text{vect}(\mathbf{W}); \mathbf{b}]$. Ahora los jacobianos vuelven a ser matrices corrientes: el jacobiano de entrada, de forma (nc', nc) , y el jacobiano de los pesos, de forma (nc', cc') . Compruébelo con las fotos: 320 filas por 3200 columnas vuelven a ser 1 024 000, y 320 por 1000 vuelven a ser 320 000; no se perdió nada, solo se reordenó.



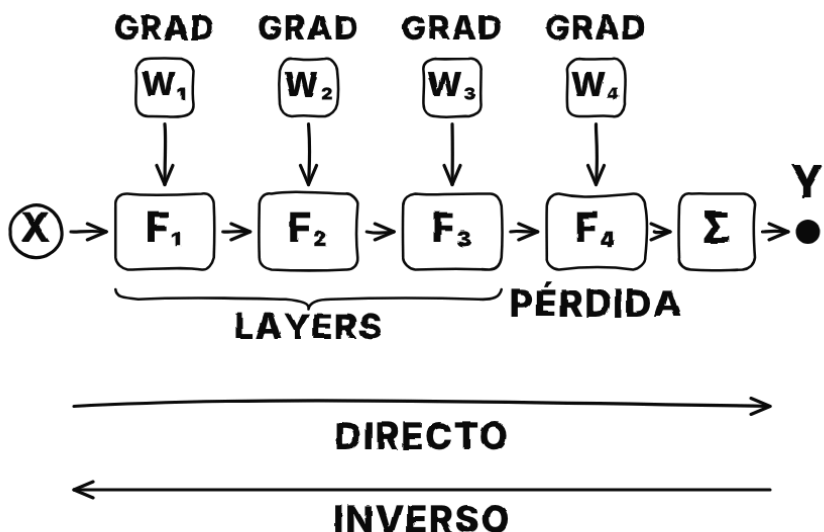
Aplanar una tabla

Hay dos consecuencias que importan para todo lo que sigue. Primera: siempre que el libro dice «el tamaño de entrada c », se refiere al producto de todas las dimensiones de la entrada, minilote incluido; para nuestra hoja eso es 3200, no 100. Segunda: un millón de casillas para una sola capa modesta es una advertencia: aunque sepamos calcular un jacobiano, quizá no queramos materializarlo, es decir, construirlo de verdad en memoria. Recuerde esa advertencia; el capítulo la convertirá en un método.

Una última nota sobre la notación. Escribir cada primitiva como una función de la entrada y los pesos coincide con bibliotecas funcionales como JAX, donde todo es una función. Los entornos orientados a objetos, como TensorFlow y PyTorch, hacen de una capa un objeto

que guarda dentro sus parámetros, con un método en lugar de la llamada a la función. Ese estilo permite la inicialización perezosa, crear los parámetros solo cuando se conoce la forma de la entrada, pero añade una capa de abstracción entre las fórmulas del libro y el código que funciona, y los dos estilos implementan además la diferenciación automática de forma distinta.

Ahora ya se puede enunciar la tarea. Tome un programa de l primitivas en fila: $\mathbf{h}_1 = f_1(\mathbf{x}, \mathbf{w}_1)$, $\mathbf{h}_2 = f_2(\mathbf{h}_1, \mathbf{w}_2)$, y así sucesivamente, hasta $\mathbf{h}_l = f_l(\mathbf{h}_{l-1}, \mathbf{w}_l)$, y por último $y = \Sigma \mathbf{h}_l$. Esta lista se llama la traza de evaluación del programa. A grandes rasgos, los primeros $l - 1$ pasos son las capas del modelo, el paso l es la pérdida de cada ejemplo, como la entropía cruzada, y la suma final reúne las pérdidas de todo el minilote. Así que el programa termina siempre en un único número, un escalar, porque la optimización numérica necesita un número que hacer más pequeño; el libro llama $F(\mathbf{x})$ a todo el programa. Con tres capas, l es 4: tres capas, un paso de pérdida y luego la suma.



La traza de evaluación

Sigue la definición. La diferenciación automática es la tarea de calcular, de forma simultánea y eficiente, todos los jacobianos de los pesos del programa, conociendo su grafo y los jacobianos de entrada y de los pesos de cada primitiva: $AD(F(x))$ es el conjunto de las $\partial_{w_i} y$, con $i = 1, \dots, l$. Como y es un solo número, cada uno de ellos es tan grande como sus pesos: una casilla por peso, que dice cómo se mueve la pérdida cuando se empuja ese peso. Hay dos familias de algoritmos, el modo directo y el modo inverso, que solo se diferencian en el orden en que se multiplican las piezas individuales. El libro anuncia que el modo inverso, llamado retropropagación en las redes neuronales, es aquí mucho más eficiente, y que la deducción se extiende a grafos ramificados y a la compartición de pesos, en la que un mismo conjunto de pesos lo usan varias capas, como en el capítulo 13.

Para recordar

Las primitivas trabajan a menudo con arreglos de rango mayor que dos, así que sus jacobianos tendrían rango 4; aplanar cada arreglo en un solo vector largo los convierte de nuevo en matrices corrientes. A partir de ahora, «el tamaño de entrada» significa el producto de todas las dimensiones de la entrada, minilote incluido, y los números crecen deprisa: cerca de un millón de casillas para una capa pequeña sobre 32 fotos. Ese tamaño es la razón por la que quizá no queramos construir los jacobianos en memoria, aunque podamos calcularlos. Un programa es una traza de evaluación de l primitivas que termina en una suma, así que su salida y es siempre un único número. La diferenciación automática consiste en calcular a la vez y de forma barata todos los gradientes de los pesos de y; el modo directo y el inverso solo se diferencian en el orden de las multiplicaciones, y gana el inverso.

Preguntas

1. ¿Por qué aplanamos el libro $\mathbf{X}$ en $\text{vect}(\mathbf{X})$ y los pesos en $[\text{vect}(\mathbf{W}); \mathbf{b}]$?
 - a. Para reducir el número de derivadas parciales de la capa
 - b. Para convertir los jacobianos de rango 4 de nuevo en matrices corrientes sin perder ninguna casilla
 - c. Porque así se puede ignorar la dimensión del minilote

- d. Porque las bibliotecas funcionales no pueden guardar matrices
2. Una capa trabaja con un minilote de 32 fotos de 100 números cada una. Cuando el libro habla de «el tamaño de entrada c », ¿cuánto vale c aquí?
- a. 100, los números por foto
 - b. 32, el tamaño del minilote
 - c. 3200, el producto de todas las dimensiones de la entrada
 - d. 132, la suma de las dos dimensiones
3. ¿Por qué el programa de la traza de evaluación termina siempre en un único número y?
- a. Porque cada capa reduce su entrada a un solo número
 - b. Porque la optimización numérica necesita un número que hacer más pequeño, así que se suman las pérdidas por ejemplo
 - c. Porque de otro modo no se podrían aplanar los jacobianos
 - d. Porque la diferenciación en modo directo solo funciona con escalares
4. ¿Qué separa la diferenciación automática en modo directo de la del modo inverso, tal como las anuncia el libro?
- a. El orden en que se multiplican las piezas individuales
 - b. El modo directo usa diferenciación numérica y el inverso, simbólica
 - c. El modo directo necesita los jacobianos de entrada, el inverso solo los de los pesos

- d. El modo inverso solo funciona en entornos orientados a objetos

Respuestas

1. **b)** Aplanar solo reordena los números: los arreglos de rango 4 se convierten en matrices de forma (nc', nc) y (nc', cc') con exactamente el mismo número de casillas.
2. **c)** El tamaño de entrada es el producto de todas las dimensiones de la entrada, minilote incluido: 32 por 100 son 3200.
3. **b)** Los últimos pasos calculan una pérdida por ejemplo y luego la suman en todo el minilote, lo que da el escalar que el entrenamiento hace más pequeño.
4. **a)** Los dos calculan los mismos gradientes de los pesos; solo se diferencian en el orden de composición, y el modo inverso resulta mucho más eficiente aquí.

Ejemplos resueltos

Contar las casillas del jacobiano de una capa Una capa totalmente conectada $f(\mathbf{X}, \mathbf{W}) = \mathbf{XW} + \mathbf{b}$ trabaja con un minilote de $n = 8$ ejemplos de $c = 4$ números cada uno y da $c' = 3$ salidas por ejemplo. ¿Cuántas casillas tiene su jacobiano de entrada, y qué forma toma después de aplanarlo?

Antes de aplanar, el jacobiano de entrada empareja cada número de salida con cada número de entrada, así que su forma es $(n, c', n, c) = (8, 3, 8, 4)$.

Sus casillas: $8 \times 3 \times 8 \times 4 = 768$.

Después de aplanar, la salida es un vector de $n \cdot c' = 24$ números y la entrada uno de $n \cdot c = 32$ números, así que el jacobiano es una matriz de forma $(24, 32)$.

Comprobación: $24 \times 32 = 768$, la misma cuenta, porque aplanar solo reordena los números.

Respuesta: 768 casillas; forma aplanada (24, 32).

Ejercicios

1. Una capa trabaja con $n = 16$ ejemplos de $c = 50$ números cada uno y da $c' = 5$ salidas por ejemplo. Dé el tamaño de entrada en el sentido del libro, y las formas aplanadas del jacobiano de entrada y del jacobiano de los pesos (con **W** de forma (c, c')).

Pista: El tamaño de entrada es el producto de todas las dimensiones de la entrada; los jacobianos aplanados son (nc', nc) y (nc', cc') .

Respuesta: Tamaño de entrada $16 \times 50 = 800$. Jacobiano de entrada (80, 800), es decir, 64 000 casillas; jacobiano de los pesos (80, 250), es decir, 20 000 casillas.

2. Un modelo tiene 5 capas, seguidas de una pérdida por ejemplo y una suma sobre el minilote. Escriba cómo termina su traza de evaluación y diga cuánto vale l .

Pista: Cuente las primitivas antes de la suma final: las capas y el paso de pérdida.

Respuesta: La traza va de $\mathbf{h}_1 = f_1(\mathbf{x}, \mathbf{w}_1)$ hasta $\mathbf{h}_6 = f_6(\mathbf{h}_5, \mathbf{w}_6)$, siendo el sexto paso la pérdida, y termina con $y = \Sigma \mathbf{h}_6$. Así que l vale 6, e y es un único número.

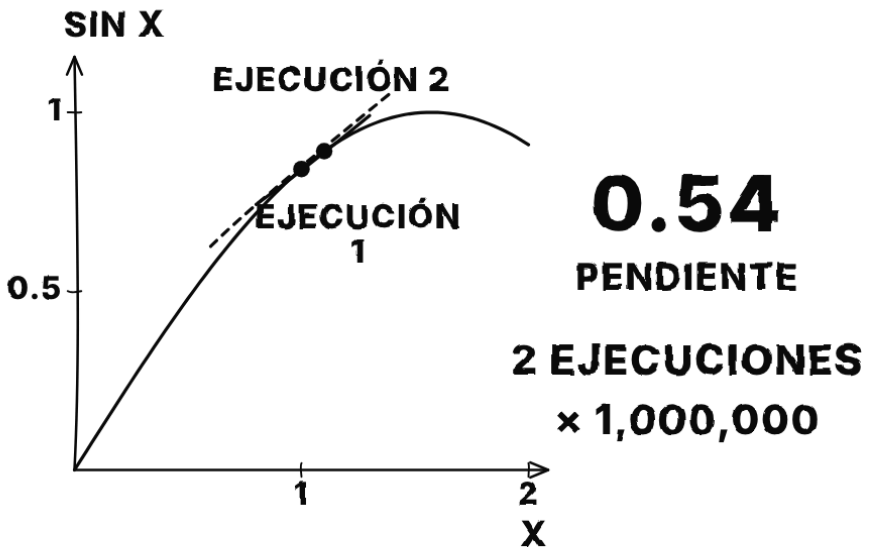
3. Explique con sus propias palabras por qué el libro advierte de que quizá no queramos materializar los jacobianos aunque sepamos calcularlos.

Pista: Piense en la cuenta que sale para una sola capa modesta sobre un minilote.

Respuesta: En cuanto el tamaño de entrada incluye el minilote, los jacobianos aplanados se vuelven enormes: cerca de un millón de casillas para una capa pequeña sobre 32 fotos. Construir tablas así en memoria para cada capa desperdiciaría espacio, así que el método debe obtener los gradientes sin guardarlas nunca enteras.

Tres formas de obtener una derivada

Tome una calculadora de bolsillo y la función seno. Calcule el seno de 1, luego el seno de 1,001, réstelos y divida entre 0,001. Obtiene unos 0,54, y el coseno de 1, la derivada verdadera, es 0,5403. Funciona, y costó dos ejecuciones de la función obtener una derivada. Retenga esas dos ejecuciones, porque deciden la suerte de este primer método.

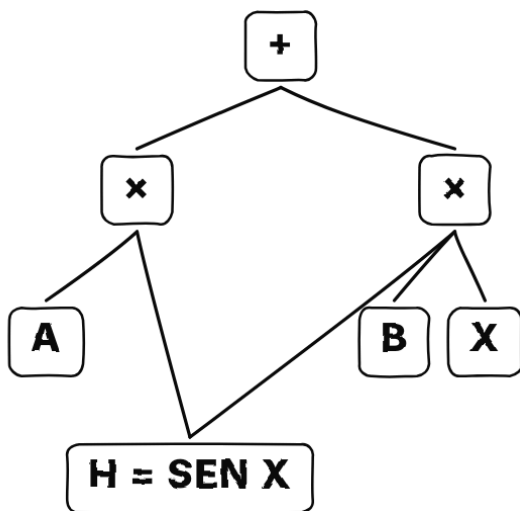


Una pendiente a partir de dos ejecuciones

Lo que acaba de hacer es diferenciación numérica: aplicar directamente la definición de derivada, con un empujón pequeño, para obtener una aproximación. El libro señala su coste: en una

implementación ingenua, cada número que se quiera derivar necesita dos llamadas a la función. Un modelo con un millón de pesos necesitaría dos millones de ejecuciones del programa entero para un gradiente, y un asistente de chat tiene muchísimos más pesos. Así que el autor reserva la diferenciación numérica para una sola tarea: comprobar, de vez en cuando, que una implementación más rápida da los números correctos.

La segunda forma es la diferenciación simbólica. Tome la pequeña función del libro $f(x) = a \sin(x) + b x \sin(x)$. Un motor simbólico, un programa de álgebra, puede calcular de antemano la fórmula completa de la derivada, y el libro lo muestra con la biblioteca de Python SymPy: se declaran x , a y b como símbolos, se escribe la función, se pide su derivada y se imprime el resultado. Ese resultado es $f'(x) = a \cos(x) + b x \cos(x) + b \sin(x)$. ¿De dónde sale? El primer término convierte el seno en coseno; el segundo es un producto, x por seno, así que da dos términos, uno por cada factor. Compruébelo con números: tome a igual a 2, b igual a 3 y x igual a $\pi/2$, donde el seno vale 1 y el coseno 0; la derivada es 2 por 0, más 3 por $\pi/2$ por 0, más 3 por 1, es decir, 3.



NP-COMPLETO

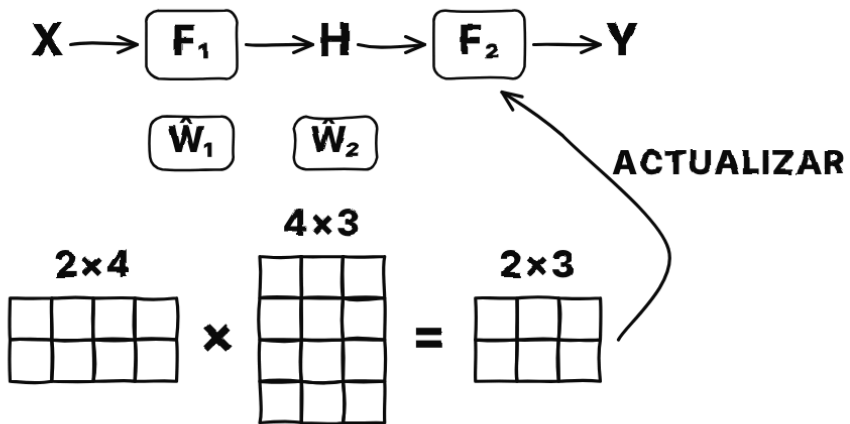
GUARDAR, REUSAR

El mismo seno calculado dos veces

Mire otra vez la fórmula y verá repeticiones: $\sin(x)$ aparece dos veces en la función y $\cos(x)$ dos veces en la derivada. Un programa sensato calcula $h = \sin(x)$ una vez, lo guarda y lo reutiliza, también en el cálculo de la derivada, y lo mismo con el coseno. Suena a simple orden, pero el autor advierte que es menos trivial de lo que parece: encontrar la mejor forma posible de calcular un jacobiano, sin ningún paso innecesario, es un problema llamado acumulación óptima del jacobiano, y es NP-completo, lo que significa que no se conoce ningún método general rápido para resolverlo. La buena noticia es que un programa tiene una estructura que podemos aprovechar. La diferenciación automática la usa para ser mucho mejor que el enfoque simbólico sin más; el libro añade que, de hecho, equivale a un enfoque

simbólico al que se le permite guardar y reutilizar esas piezas intermedias.

Ahora, la tercera forma, partiendo de la regla de la cadena. Tome dos primitivas seguidas, $\mathbf{h} = f_1(\mathbf{x})$, $\mathbf{y} = f_2(\mathbf{h})$. La regla de la cadena de los jacobianos dice $\partial_{\mathbf{x}}\mathbf{y} = \partial_{\mathbf{h}}\mathbf{y} \cdot \partial_{\mathbf{x}}\mathbf{h}$. Si $\mathbf{x}$, $\mathbf{h}$ y $\mathbf{y}$ tienen a , b y c componentes, eso es una matriz de c por b por una matriz de b por a , y el resultado es de c por a , una fila por salida y una columna por entrada, como debe ser. Con 3 entradas, 4 valores intermedios y 2 salidas, una tabla de 2 por 4 por una tabla de 4 por 3 da una tabla de 2 por 3. El libro lee la regla como una actualización: una vez que tiene f_1 y su jacobiano, aplicar f_2 significa multiplicar lo que tiene por el jacobiano de f_2 , y vuelve a estar al día.

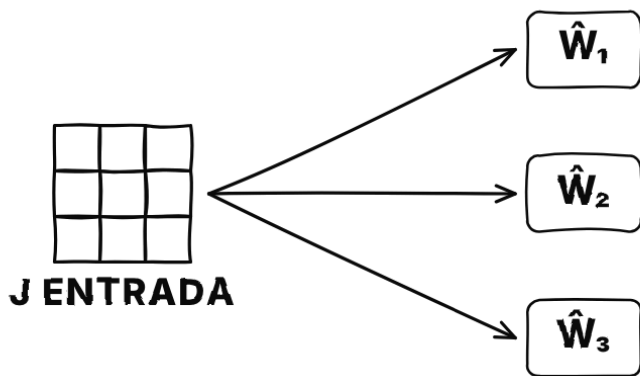


La regla de la cadena como producto de tablas

Esa lectura ya es un algoritmo, la diferenciación automática en modo directo, F-AD para abreviar. Cada vez que se ejecuta una primitiva, empieza una tabla nueva para sus pesos, llamada tangente, y actualiza todas las tangentes que ya tiene. En el primer paso, $\mathbf{h}_1 = f_1(\mathbf{x}, \mathbf{w}_1)$, todavía no hay nada guardado, así que la tangente es simplemente el jacobiano de los pesos: $\hat{\mathbf{W}}_1 = \partial_{\mathbf{w}_1} \mathbf{h}_1$. En el segundo paso, $\mathbf{h}_2 = f_2(\mathbf{h}_1, \mathbf{w}_2)$, la tangente vieja se multiplica por el jacobiano de entrada de f_2 , $\hat{\mathbf{W}}_1 \leftarrow [\partial_{\mathbf{h}_1} \mathbf{h}_2] \hat{\mathbf{W}}_1$, y empieza una nueva, $\hat{\mathbf{W}}_2 = \partial_{\mathbf{w}_2} \mathbf{h}_2$.

Pruébalo

Pruébalo con números sueltos en una calculadora. Sea la entrada x igual a 2, que el primer paso multiplique por un peso w_1 igual a 3, y que el segundo multiplique por un peso w_2 igual a 5. Tras el primer paso, la tangente es la derivada de 3 por 2 respecto de w_1 , que es x , o sea 2. En el segundo paso, el jacobiano de entrada es w_2 , 5, así que la tangente pasa a ser 5 por 2, que es 10; la tangente nueva para w_2 es el valor que entra, 6. Compruébalo ahora a mano: la salida es w_2 por w_1 por x , su derivada respecto de w_1 es 5 por 2, que es 10, y respecto de w_2 es 3 por 2, que es 6.



El modo directo en una calculadora

Para la i -ésima primitiva en general, $\mathbf{h}_i = f_i(\mathbf{h}_{i-1}, \mathbf{w}_i)$, la regla es la misma: $\hat{W}_j \leftarrow [\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i] \hat{W}_j$ para todo $j < i$, y $\hat{W}_i = \partial_{\mathbf{w}_i} \mathbf{h}_i$. Así que la actualización usa el jacobiano de entrada del paso, y la tangente nueva, su jacobiano de los pesos. En el paso i hay $i - 1$ actualizaciones, una por cada tangente en memoria, y todas comparten el mismo jacobiano de entrada; en el cuarto paso se actualizan tres tangentes con una misma tabla. La última operación del programa es la suma, y su gradiente dará la salida del algoritmo.

Para recordar

La diferenciación numérica empuja cada número y ejecuta la función dos veces, lo que está bien para comprobar pero no tiene remedio con millones de pesos. La diferenciación simbólica escribe la fórmula de la derivada, pero repite trabajo, y encontrar la forma perfectamente económica de calcular un jacobiano es un problema NP-completo. La diferenciación automática usa en cambio la estructura del programa, como un método simbólico que puede guardar y reutilizar valores intermedios. El modo directo lee la regla de la cadena como una actualización: en cada paso, una tangente nueva parte del jacobiano de los pesos, y cada tangente anterior se multiplica por el jacobiano de entrada del paso.

Preguntas

1. ¿Por qué reserva el libro la diferenciación numérica solo para comprobar una implementación?
 - a. Porque da el signo equivocado en la mayoría de las derivadas
 - b. Porque cada número que hay que derivar necesita dos llamadas a la función entera, demasiadas para millones de pesos
 - c. Porque no puede tratar la función seno
 - d. Porque necesita un motor simbólico para funcionar

2. ¿Qué problema tiene limitarse a escribir simbólicamente la fórmula de la derivada?
 - a. La fórmula suele estar mal en los productos
 - b. Los motores simbólicos no saben derivar el seno
 - c. Repite trabajo, y encontrar la forma perfectamente económica de calcular un jacobiano es NP-completo
 - d. Necesita dos llamadas a la función por número, como la diferenciación numérica

3. En el modo directo, ¿qué les pasa a las tangentes que ya están en memoria cuando se ejecuta la i -ésima primitiva?
 - a. Cada una se multiplica por el jacobiano de entrada de la i -ésima primitiva, una sola tabla compartida por todas
 - b. Cada una se multiplica por el jacobiano de los pesos de la i -ésima primitiva
 - c. Se descartan y se recalculan desde el principio
 - d. Se quedan igual hasta la suma final

Respuestas

1. **b)** En una implementación ingenua, cada escalar que hay que derivar cuesta dos llamadas a la función, así que un modelo con un millón de pesos necesitaría dos millones de ejecuciones para un gradiente.

2. **c)** Piezas como $\sin(x)$ y $\cos(x)$ aparecen varias veces; evitar en general todo cálculo innecesario es el problema NP-completo de la acumulación óptima del jacobiano.

3. **a)** La regla $\hat{W}_j \leftarrow [\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i] \hat{W}_j$ para todo $j < i$ actualiza cada tangente anterior con el mismo jacobiano de entrada, mientras que la tangente nueva parte del jacobiano de los pesos.

Ejemplos resueltos

El ejemplo simbólico del libro, en un punto Para la función del libro $f(x) = a \sin(x) + b x \sin(x)$, el motor simbólico devuelve $f'(x) = a \cos(x) + b x \cos(x) + b \sin(x)$. Evalúe la derivada en $x = 0$.

En $x = 0$ el seno vale 0 y el coseno vale 1.

El primer término da a por 1, que es a .

El segundo término da b por 0 por 1, que es 0, por el factor x .

El tercer término da b por 0, que es 0.

Respuesta: $f'(0) = a$.

El modo directo a lo largo de tres pasos Un programa multiplica su entrada $x = 1$ por $w_1 = 2$, luego por $w_2 = 4$ y luego por $w_3 = 0,5$. Lleve las tangentes hacia delante y dé la derivada de la salida respecto de cada peso.

Paso 1: $h_1 = w_1 x = 2$; la tangente nueva es el jacobiano de los pesos, la derivada de $w_1 x$ respecto de w_1 , que es $x = 1$.

Paso 2: $h_2 = w_2 h_1 = 8$; el jacobiano de entrada es $w_2 = 4$, así que la tangente vieja pasa a ser $4 \times 1 = 4$; la tangente nueva es $h_1 = 2$.

Paso 3: $h_3 = w_3 h_2 = 4$; el jacobiano de entrada es $w_3 = 0,5$, así que las tangentes pasan a ser $0,5 \times 4 = 2$ y $0,5 \times 2 = 1$; la tangente nueva es $h_2 = 8$.

Comprobación: $h_3 = w_3 w_2 w_1 x$, cuyas derivadas son $w_3 w_2 x = 2$, $w_3 w_1 x = 1$ y $w_2 w_1 x = 8$.

Respuesta: 2 para w_1 , 1 para w_2 , 8 para w_3 .

Ejercicios

1. Un modelo pequeño tiene 5000 pesos. ¿Cuántas llamadas al programa necesita la diferenciación numérica ingenua para un gradiente completo?

Pista: Dos llamadas por cada número que hay que derivar.

Respuesta: 10 000 llamadas: dos por cada uno de los 5000 pesos.

2. En $\mathbf{h} = f_1(\mathbf{x})$, $\mathbf{y} = f_2(\mathbf{h})$, la entrada tiene 5 componentes, $\mathbf{h}$ tiene 7 e $\mathbf{y}$ tiene 1. Dé las formas de los dos jacobianos de la regla de la cadena y de su producto.

Pista: Una fila por salida y una columna por entrada, en cada paso.

Respuesta: $\partial_{\mathbf{h}}\mathbf{y}$ es de 1 por 7, $\partial_{\mathbf{x}}\mathbf{h}$ es de 7 por 5, y su producto $\partial_{\mathbf{x}}\mathbf{y}$ es de 1 por 5.

3. En el modo directo, ¿cuántas actualizaciones de tangentes ocurren en la sexta primitiva, y qué jacobiano comparten?

Pista: Cuente las tangentes que ya están en memoria antes del sexto paso.

Respuesta: Cinco actualizaciones, una por cada una de las tangentes de $\mathbf{w}_1$ a $\mathbf{w}_5$, todas con el mismo jacobiano de entrada de la sexta primitiva; después, la sexta tangente parte de su jacobiano de los pesos.

Darle la vuelta a la cadena

Tome un lápiz y dos tablas pequeñas de números, de 3 por 3 cada una, y multiplíquelas. Cada casilla del resultado necesita 3 multiplicaciones, y hay 9 casillas, así que hace 27 multiplicaciones. Ahora multiplique una sola fila de 3 números por una de esas tablas. El resultado es una fila de 3 casillas, de 3 multiplicaciones cada una: 9 en total. Aquí una fila por una tabla es tres veces más barata, y con tablas de mil filas la diferencia se vuelve de mil veces; esa diferencia es toda la historia de esta parte.

$$\begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} \times \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} \Rightarrow \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} \quad 27$$

$$\begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} \times \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} \Rightarrow \begin{array}{|c|c|c|} \hline & & \\ \hline & & \\ \hline & & \\ \hline \end{array} \quad 9$$

$$\begin{array}{c} \mathbf{1}^T \\ \boxed{1} \quad \boxed{1} \end{array} \times \begin{array}{c} \hat{\mathbf{W}} \\ \boxed{4} \\ \boxed{6} \end{array} = \boxed{10}$$

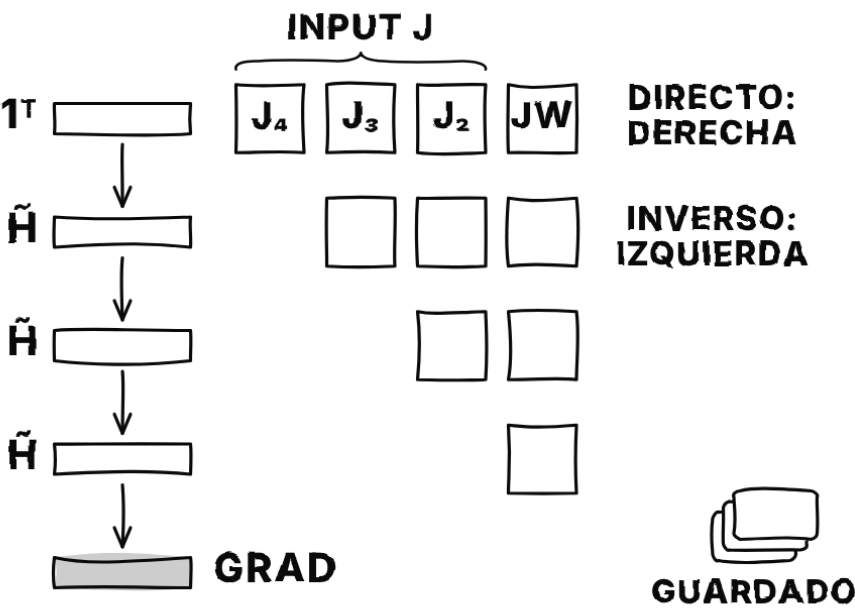
Tabla por tabla, fila por tabla

Primero, el modo directo termina su trabajo. El programa termina con la suma de $\mathbf{h}_i$, así que el gradiente de cada peso sale de la última tangente: $\nabla_{\mathbf{w}_i} y = \mathbf{1}^T \hat{\mathbf{W}}_i$ para todo i . En palabras: $\mathbf{1}$ es un vector lleno de unos, y multiplicar por él suma las filas de la tangente, que es exactamente lo que la suma final hace con las salidas. Con dos salidas y una tangente cuyas filas son 4 y 6, el gradiente es 4 más 6, que es 10. Una nota en letra pequeña del libro: esto da una fila en lugar de la columna con que se definieron los gradientes; el autor lo deja de lado hasta más adelante en el capítulo.

¿Qué tal es el modo directo? En el lado bueno, cada actualización puede intercalarse con el programa original, paso a paso, así que la memoria que necesita es más o menos la que necesita el propio programa. En el lado malo, su movimiento central, actualizar una tangente, multiplica dos matrices, de formas (c'_i, c_i) y (c_i, p_j) , donde p_j es el número de pesos de $\mathbf{w}_j$. Si entradas y salidas tienen ambas forma (n, d) , con n ejemplos en el minilote y d características cada uno, el coste es $\mathcal{O}(n^2 d^2 p_j)$: crece con el cuadrado del lote y con el cuadrado de las características. Con 32 fotos, 100 características y 10 000 pesos, eso es 1024 por 10 000 por 10 000, unos cien mil millones de multiplicaciones, para una actualización de una tangente. Para imágenes, con muchas más características, se vuelve inviable enseguida.

La salida está escondida en el último paso del modo directo, que no es una matriz por una matriz sino un vector por una matriz, porque la salida es un único número. Para aprovecharlo, el libro escribe entero el gradiente de un peso: $\nabla_{\mathbf{w}_i} y = \mathbf{1}^T [\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i] \cdots [\partial_{\mathbf{h}_i} \mathbf{h}_{i+1}] [\partial_{\mathbf{w}_i} \mathbf{h}_i]$. Notación aparte, no es más que una larga fila de multiplicaciones: el vector de unos, luego una cadena de jacobianos de entrada, que el libro imprime en rojo, y en el extremo derecho el jacobiano de los

pesos del paso i , en verde. La parte roja recibe un nombre propio: $\tilde{\mathbf{h}}_i = \mathbf{1}^T \prod_{j=i+1}^J \partial_{\mathbf{h}_{j-1}} \mathbf{h}_j$.



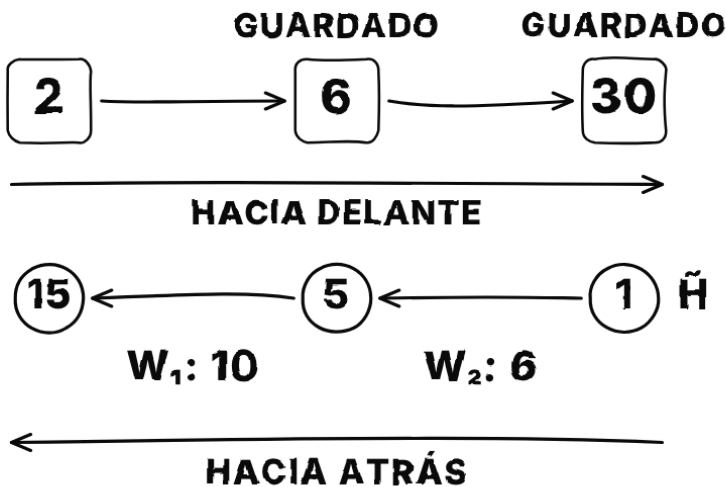
Multiplicar desde la izquierda

La multiplicación de matrices es asociativa: puede agrupar el producto como quiera y el resultado es el mismo. El modo directo multiplicaba de derecha a izquierda, en el orden en que se ejecutaba el programa, siempre con una matriz en la mano. Si en cambio multiplica desde la izquierda, ocurren dos cosas. La pieza de más a la izquierda es un vector por una matriz, barato, y su resultado es otra vez un vector, así que lo barato nunca se acaba. Y $\tilde{\mathbf{h}}_i$ puede construirse de forma recursiva, empezando por el último paso y multiplicando por los jacobianos de entrada en orden inverso.

Esto da la diferenciación automática en modo inverso, R-AD, en tres pasos. Primero, ejecute todo el programa y guarde cada salida intermedia. Segundo, empiece con el vector $\tilde{\mathbf{h}} = \mathbf{1}^T$, el término de más a la izquierda. Tercero, vuelva hacia atrás, con i bajando de l a 1 , y en cada paso haga dos cosas: obtenga el gradiente de los pesos de ese paso, $\partial_{\mathbf{w}_i} y = \tilde{\mathbf{h}} [\partial_{\mathbf{w}_i} \mathbf{h}_i]$, y luego pase el vector un paso atrás, $\tilde{\mathbf{h}} \leftarrow \tilde{\mathbf{h}} [\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i]$. Las dos cosas son un vector por una matriz.

Pruébalo

Retome la cadena de la calculadora de la parte anterior: x es 2, multiplicado por w_1 igual a 3, y luego por w_2 igual a 5. Ejecútela hacia delante y anote lo que ve: 6 tras el primer paso, 30 tras el segundo. Ahora empiece hacia atrás con $\tilde{h}$ igual a 1. En el paso 2, el gradiente de w_2 es 1 por el valor que entró, 6; luego $\tilde{h}$ pasa a ser 1 por w_2 , que es 5. En el paso 1, el gradiente de w_1 es 5 por x , que es 10; obtiene los mismos 10 y 6 que en el modo directo, y el último $\tilde{h}$, 15, es la derivada de la salida respecto de x .



Hacia atrás en la calculadora

Esos tres pasos forman un programa que es aproximadamente un reflejo del original, y que el libro llama programa dual o inverso. Los vectores $\mathbf{h}$ se llaman adjuntos: uno tras otro contienen el gradiente de la salida respecto de cada $\mathbf{h}_i$, mientras que las tangentes del modo directo contenían los gradientes de cada $\mathbf{h}_i$ respecto de los pesos. En el lenguaje de las redes neuronales, el programa original o primal es la pasada hacia delante, que no es lo mismo que el modo directo, y el programa inverso es la pasada hacia atrás. A diferencia del modo directo, toda la pasada hacia delante debe terminar antes de que pueda empezar la pasada hacia atrás, y los entornos necesitan una maquinaria especial para guardar todas las salidas intermedias, cada uno a su manera. Entrenar una aplicación de fotos funciona

exactamente así: las fotos atraviesan la red, se calcula la pérdida, y luego los gradientes fluyen hacia atrás.

Para recordar

El modo directo termina con $\nabla_{\mathbf{w}_i} y = \mathbf{1}^T \hat{\mathbf{W}}_i$ y usa poca memoria, pero cada actualización de tangente multiplica dos matrices, con un coste del orden de $n^2 d^2 p_j$ que se dispara con las imágenes. Escrito entero, cada gradiente es una fila de multiplicaciones que empieza con el vector de unos; como el orden de multiplicar es libre, empezar por la izquierda hace que cada producto sea un vector por una matriz. El modo inverso ejecuta primero el programa, guardando cada valor intermedio, y luego vuelve hacia atrás llevando el adjunto $\mathbf{h}$, lo que da un gradiente de pesos por paso. La pasada hacia delante debe terminar antes de que empiece la pasada hacia atrás, y por eso hay que guardar todo lo que hay entre medias.

Preguntas

1. ¿Por qué el modo directo se vuelve inviable para entradas como las imágenes, aunque necesite poca memoria?
 - a. Porque debe guardar cada salida intermedia del programa
 - b. Porque cada actualización de tangente multiplica dos matrices, con un coste que crece como $n^2 d^2 p_j$
 - c. Porque no puede intercalarse con el programa original
 - d. Porque las imágenes no tienen jacobiano de los pesos

2. ¿Qué permite calcular el mismo producto de jacobianos desde la izquierda en lugar de desde la derecha?
 - a. La multiplicación de matrices es asociativa, así que los paréntesis pueden ponerse libremente
 - b. Los jacobianos son siempre matrices cuadradas
 - c. Los jacobianos de entrada son todos iguales entre sí
 - d. El jacobiano de los pesos es un vector de unos
3. ¿Por qué el modo inverso debe ejecutar todo el programa antes de que pueda empezar la pasada hacia atrás?
 - a. Porque la pasada hacia atrás solo necesita el valor de la pérdida al final del todo
 - b. Porque la pasada hacia atrás empieza en la salida y necesita las salidas intermedias guardadas de cada paso
 - c. Porque los pesos cambian durante la pasada hacia delante
 - d. Porque el vector de unos solo se conoce después de ejecutar el programa
4. ¿Qué contiene un adjunto $\mathbf{h}_i$, en comparación con una tangente del modo directo?
 - a. El gradiente de $\mathbf{h}_i$ respecto de los pesos, como una tangente
 - b. El propio valor de $\mathbf{h}_i$, guardado durante la pasada hacia delante
 - c. El gradiente de la salida respecto de $\mathbf{h}_i$, mientras que una tangente contiene el gradiente de $\mathbf{h}_i$ respecto de los pesos
 - d. El jacobiano de los pesos del paso i

Respuestas

1. **b)** La actualización central multiplica matrices de formas (c'_i, c_i) y (c_i, p_j) ; con entradas y salidas de forma (n, d) , eso cuesta del orden de $n^2 d^2 p_j$, cuadrático en el tamaño del lote y en las características.
2. **a)** Como la multiplicación de matrices es asociativa, el producto largo puede hacerse en cualquier orden; empezar por el vector de unos hace que cada paso sea un vector por una matriz.
3. **b)** El adjunto empieza en la salida y recorre hacia atrás cada paso, usando jacobianos que dependen de los valores intermedios, así que todos ellos deben haberse calculado y guardado antes.
4. **c)** Los adjuntos guardan, paso a paso, los gradientes de la salida respecto de cada valor intermedio; las tangentes iban en sentido contrario, de los pesos a los valores intermedios.

Ejemplos resueltos

Una pasada hacia atrás por tres pasos Un programa multiplica $x = 1$ por $w_1 = 2$, luego por $w_2 = 4$ y luego por $w_3 = 0,5$, y su salida es el resultado. Use el modo inverso para obtener el gradiente de cada peso.

Pasada hacia delante, guardándolo todo: $h_1 = 2$, $h_2 = 8$, $h_3 = 4$.

Empiece el adjunto en $\tilde{h} = 1$.

Paso 3: el gradiente de w_3 es $\tilde{h}$ por el valor que entró, $1 \times 8 = 8$; luego $\tilde{h}$ pasa a ser $1 \times w_3 = 0,5$.

Paso 2: el gradiente de w_2 es $0,5 \times h_1 = 1$; luego $\tilde{h}$ pasa a ser $0,5 \times w_2 = 2$.

Paso 1: el gradiente de w_1 es $2 \times x = 2$; el último $\tilde{h}$, $2 \times w_1 = 4$, es la derivada de la salida respecto de x .

Respuesta: 8 para w_3 , 1 para w_2 , 2 para w_1 .

Ejercicios

1. Una tangente tiene dos filas, (1, 2) y (3, 4). ¿Qué da $\mathbf{1}^T$ por esta tangente?

Pista: Multiplicar por un vector de unos suma las filas.

Respuesta: La fila (4, 6): $1 + 3$ y $2 + 4$.

2. La actualización del modo directo cuesta del orden de $n^2 d^2 p_j$ multiplicaciones. Estímela para $n = 10$ ejemplos, $d = 20$ características y $p_j = 50$ pesos, y luego diga qué ocurre cuando el lote se duplica.

Pista: Eleve al cuadrado n y d , y luego multiplique por p_j .

Respuesta: $100 \times 400 \times 50 = 2\,000\,000$ de multiplicaciones. Duplicar el lote multiplica el coste por cuatro, porque crece con n al cuadrado.

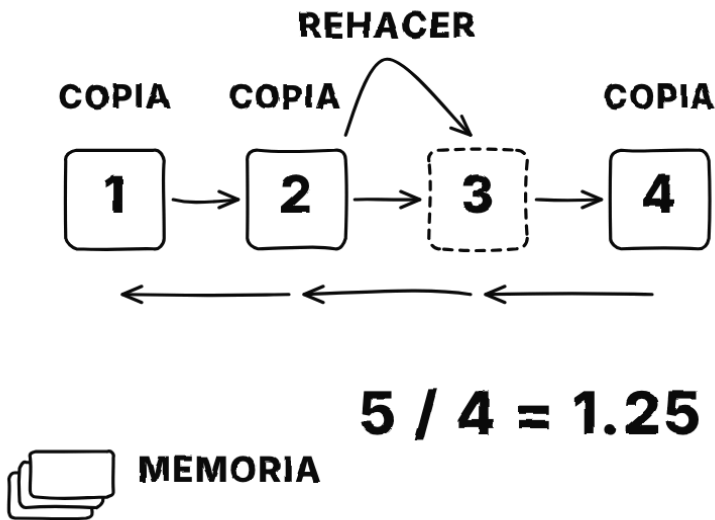
3. Explique por qué llevar un vector desde la izquierda es más barato que llevar una matriz desde la derecha, para un programa cuya salida es un único número.

Pista: Compare un vector por una tabla con una tabla por una tabla.

Respuesta: Como la salida es un número, el término más a la izquierda es un vector, y un vector por una matriz da otro vector; cada paso sigue siendo un producto vector-matriz. Desde la derecha, cada paso multiplica dos matrices, lo que con tablas de mil filas cuesta unas mil veces más.

Ahorrar memoria y pedir solo productos

Piense en un nivel largo de un videojuego con puntos de guardado. Si pudiera guardar después de cada sala, un error no le costaría nada, pero su tarjeta de memoria se llenaría enseguida. Si guarda solo cada pocas salas, la tarjeta se mantiene ligera, y tras un error repite unas cuantas salas desde el último guardado. Los jugadores hacen este intercambio sin pensarlo. El modo inverso tiene que hacer exactamente el mismo intercambio, y la solución del libro hasta lleva el nombre que usan los jugadores: puntos de control.



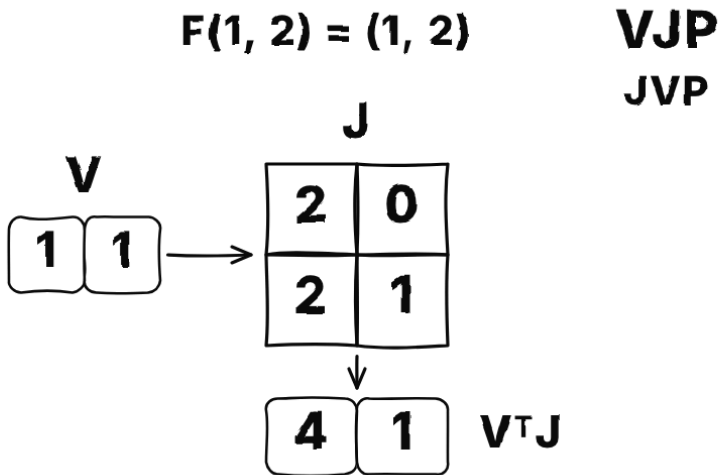
Puntos de control en cuatro bloques

Primero, la buena noticia, en números. El modo inverso es mucho más eficiente que el directo, porque los dos movimientos de su paso hacia atrás son un vector por una matriz, y ese coste crece solo linealmente con todos los tamaños en juego. Duplique el lote y el trabajo se duplica, donde el del modo directo se cuadruplica. El precio es la memoria: cada valor intermedio del programa primal tiene que guardarse para la pasada hacia atrás, y el libro dice que hay que guardarlo con una estrategia adecuada, incluso en disco. En una red entrenada con fotos de alta resolución, esos valores guardados pueden ocupar mucho más que los propios pesos.

Los puntos de control del gradiente mejoran el intercambio gastando algo de cálculo extra para ahorrar parte de la memoria. Solo se guardan unas pocas salidas intermedias, los puntos de control, y las demás se recalculan durante la pasada hacia atrás cuando hacen falta. La figura del libro lo muestra en cuatro paneles, con un modelo de cuatro bloques dibujados en fila. En el primero se ejecuta una pasada hacia delante, pero solo se guardan las salidas del primer, segundo y cuarto bloque. En el segundo, la pasada hacia atrás, dibujada con flechas rojas de derecha a izquierda, se detiene en el tercer bloque, porque sus valores no se guardaron. En el tercero, una segunda pasada hacia delante parte del punto de control más cercano, el segundo bloque, y vuelve a calcular los valores del tercer bloque; en el cuarto, la pasada hacia atrás se completa. Cuente el trabajo: cuatro bloques hacia delante más un bloque recalculado son cinco cálculos de bloque en lugar de cuatro, es decir, 1,25 veces más, la cifra que da el libro. Cuantos menos puntos de control guarde, más cuesta la pasada hacia atrás.

Mire ahora otra vez el tercer paso del modo inverso. La única operación que necesita es un vector fila v multiplicado por un

jacobiano de f , ya sea el jacobiano de entrada o el de los pesos. Estas dos operaciones se llaman los productos vector-jacobiano de f , VJP para abreviar. El libro define uno formalmente, con una traspuesta para poner el vector en el sentido correcto: para $y = f(x)$, con $x \sim (c)$ e $y \sim (c')$, es $\text{vjp}_f(v) = v^T \partial f(x)$, donde v tiene c' componentes, una por salida. En palabras: usted le da a la función un vector del tamaño de su salida, y ella le devuelve un vector del tamaño de su entrada, como si se hubiera usado el jacobiano, sin enseñarle nunca el jacobiano. Una función con varios argumentos tiene un VJP por argumento.

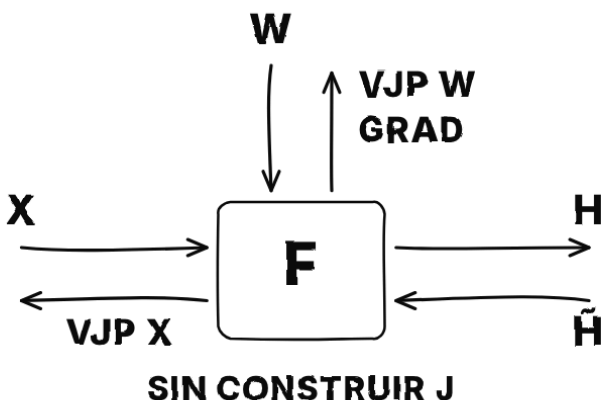


Un vector por el jacobiano

Un pequeño ejemplo para ver los números. Sea f la función que lleva $x = (1, 2)$ a $(x_1^2, x_1 x_2)$, es decir, a $(1, 2)$. Su jacobiano tiene las filas $(2x_1, 0)$ y (x_2, x_1) , es decir, $(2, 0)$ y $(2, 1)$. Con $v = (1, 1)$, el VJP es 1 por la

primera fila más 1 por la segunda: (4, 1). En cambio, el libro señala de pasada que el modo directo puede escribirse entero con la operación traspuesta, el producto jacobiano-vector, JVP, que para una sola salida es la derivada direccional vista antes.

Una primitiva tiene dos argumentos, así que tiene dos VJP: $\text{vjp_f, x}(\mathbf{v}) = \mathbf{v}^T \partial_{\mathbf{x}} f(\mathbf{x}, \mathbf{w})$ y $\text{vjp_f, w}(\mathbf{v}) = \mathbf{v}^T \partial_{\mathbf{w}} f(\mathbf{x}, \mathbf{w})$. Los dos movimientos del paso hacia atrás se convierten en dos llamadas con el adjunto: $\partial_{\mathbf{w}} y = \text{vjp_f, w}(\mathbf{h})$, y $\mathbf{h} \leftarrow \text{vjp_f, h}(\mathbf{h})$. Así que todo un sistema de diferenciación automática puede construirse eligiendo un conjunto de primitivas y dando a cada una sus dos VJP, sin materializar ningún jacobiano en memoria en ningún momento. La segunda figura del libro dibuja una primitiva como una caja: en la pasada hacia delante entran la entrada y los pesos y sale la salida; en la pasada hacia atrás el adjunto entra por la derecha, y dos llamadas, una por VJP, envían un vector de vuelta hacia la entrada y otro a los pesos. Esto es lo que ocurre, paso tras paso, cuando el entrenamiento de un asistente de chat llama a su pasada hacia atrás.



$$E_1 \rightarrow (2, 0)$$

$$E_2 \rightarrow (2, 1)$$

Una primitiva con dos VJP

Trabajando así no se pierde nada. Llame al VJP de entrada con los vectores de la base $\mathbf{e}_1, \dots, \mathbf{e}_n$, cada uno un vector con un 1 en un lugar y 0 en los demás, y recupera el jacobiano fila a fila: $\partial_x f(\mathbf{x}, \mathbf{w}) = [\text{vjp_f}, \mathbf{x}(\mathbf{e}_1); \dots; \text{vjp_f}, \mathbf{x}(\mathbf{e}_n)]$. En nuestro pequeño ejemplo, $\mathbf{e}_1 = (1, 0)$ devuelve $(2, 0)$ y $\mathbf{e}_2 = (0, 1)$ devuelve $(2, 1)$: exactamente las dos filas. ¿Por qué merece la pena esta reformulación? El libro responde mirando las piezas de una capa totalmente conectada, proyecciones lineales y no linealidades elemento a elemento, empezando por una proyección lineal sin sesgo.

Para recordar

El modo inverso solo cuesta productos vector-matriz, lineales en cada tamaño, pero debe guardar cada valor intermedio de la pasada hacia delante. Los puntos de control del gradiente guardan solo algunos valores y recalculan el resto durante la pasada hacia atrás; en el ejemplo de cuatro bloques del libro eso cuesta 1,25 veces el cálculo. El paso hacia atrás solo necesita productos vector-jacobiano, $\text{vjp}_f(\mathbf{v}) = \mathbf{v}^T \partial f(\mathbf{x})$, uno para la entrada y otro para los pesos. Así que cada primitiva solo necesita sus dos VJP, y ningún jacobiano se construye nunca en memoria; el jacobiano completo puede recuperarse igualmente fila a fila con vectores de la base.

Preguntas

1. ¿Qué intercambian los puntos de control del gradiente, y por qué?
 - a. Gastan cálculo extra en la pasada hacia atrás para guardar menos valores intermedios
 - b. Gastan memoria extra para evitar cualquier recálculo
 - c. Sustituyen el modo inverso por el directo en los bloques del medio
 - d. Descartan algunos gradientes para ahorrar tiempo
2. En la figura del libro, ¿por qué la pasada hacia atrás cuesta 1,25 veces el cálculo habitual?

- a. Porque los cuatro bloques se ejecutan dos veces hacia delante
 - b. Porque el tercer bloque no se guardó y hay que recalcularlo una vez desde el punto de control del segundo bloque
 - c. Porque la pasada hacia atrás siempre es una cuarta parte más cara que la pasada hacia delante
 - d. Porque se perdieron los valores del cuarto bloque
3. ¿Qué le da un producto vector-jacobiano, en comparación con el propio jacobiano?
- a. El jacobiano completo, calculado más deprisa
 - b. Solo la diagonal del jacobiano
 - c. El producto $\mathbf{v}^T \partial \mathbf{f}(\mathbf{x})$, un vector del tamaño de la entrada, sin construir nunca el jacobiano
 - d. Un vector del tamaño de la salida
4. ¿Se pierde algo al dar a cada primitiva solo sus dos VJP en lugar de sus jacobianos?
- a. Sí, los jacobianos ya no pueden obtenerse de ninguna manera
 - b. No, llamar al VJP con los vectores de la base $\mathbf{e}_1, \dots, \mathbf{e}_n$ devuelve el jacobiano fila a fila
 - c. Sí, los gradientes de los pesos se vuelven aproximados
 - d. No, porque los VJP y los jacobianos tienen siempre el mismo tamaño

Respuestas

1. a) Solo se guardan unas pocas salidas, los puntos de control; las demás se recalculan desde el punto de control más cercano durante la pasada hacia atrás, lo que cuesta más cálculo pero menos memoria.

2. b) Cuatro bloques hacia delante más un bloque recalculado son cinco cálculos de bloque en lugar de cuatro, es decir, 1,25 veces más.

3. c) Usted le da a la función un vector del tamaño de su salida y recibe un vector del tamaño de su entrada, como si se hubiera usado el jacobiano.

4. b) Cada vector de la base escoge una fila, así que apilar desde $\text{vjp_f}, \mathbf{x}(\mathbf{e}_1)$ hasta $\text{vjp_f}, \mathbf{x}(\mathbf{e}_n)$ reconstruye el jacobiano de entrada.

Ejemplos resueltos

El coste del ejemplo de puntos de control del libro Un modelo tiene cuatro bloques. La pasada hacia delante solo guarda las salidas del primer, segundo y cuarto bloque. ¿Cuánto cálculo necesita la pasada hacia atrás en comparación con el caso normal, contando cálculos de bloque?

La pasada hacia delante calcula los cuatro bloques una vez: 4 cálculos de bloque.

Al volver, la pasada llega al tercer bloque y encuentra que faltan sus valores.

Vuelve a ejecutar hacia delante desde el punto de control más cercano, la salida del segundo bloque, a través del tercer bloque: 1 cálculo de bloque más.

Total 5 en lugar de 4, y $5 / 4 = 1,25$.

Respuesta: 1,25 veces el cálculo.

Un producto vector-jacobiano a mano f lleva $\mathbf{x} = (3, 4)$ a $(x_1 + x_2, x_1 x_2)$. Calcule $\text{vjp_f}(\mathbf{v})$ para $\mathbf{v} = (2, 1)$.

El jacobiano tiene una fila por salida: la primera salida da $(1, 1)$, la segunda da $(x_2, x_1) = (4, 3)$.

$\mathbf{v}^T$ por el jacobiano es 2 veces la primera fila más 1 vez la segunda.

$2 \times (1, 1) = (2, 2)$, y sumándole $(4, 3)$ da $(6, 5)$.

Respuesta: $(6, 5)$, un vector del tamaño de la entrada.

Ejercicios

1. Un modelo tiene 8 bloques. Solo las salidas de los bloques 3 y 6 no se guardan, y cada una debe recalcularse una vez durante la pasada hacia atrás. ¿Cuánto cálculo necesita la pasada hacia atrás en comparación con el caso normal?

Pista: Cuente los cálculos hacia delante y súmeles los bloques recalculados.

Respuesta: 8 cálculos hacia delante más 2 bloques recalculados son 10 en lugar de 8, es decir, 1,25 veces más.

2. f lleva $\mathbf{x}$ a $(2x_1 + x_2, x_1 - x_2)$. Llame a su VJP con $\mathbf{e}_1 = (1, 0)$ y $\mathbf{e}_2 = (0, 1)$ y escriba el jacobiano que recupera.

Pista: Cada vector de la base escoge una fila del jacobiano.

Respuesta: $\text{vjp}(\mathbf{e}_1) = (2, 1)$ y $\text{vjp}(\mathbf{e}_2) = (1, -1)$, así que el jacobiano tiene las filas $(2, 1)$ y $(1, -1)$.

3. Explique con sus propias palabras por qué el modo inverso necesita mucha más memoria que el directo, aunque calcule mucho menos.

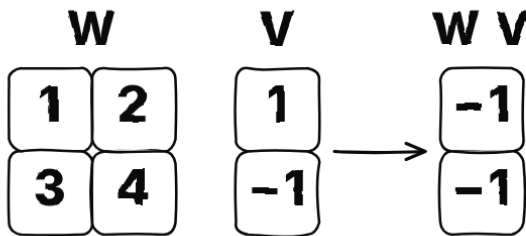
Pista: ¿Qué necesita la pasada hacia atrás de la pasada hacia delante, y cuándo?

Respuesta: El modo directo hace sus actualizaciones junto con el programa y puede olvidar cada valor una vez que ha seguido adelante. El modo inverso solo puede empezar cuando todo el programa se ha ejecutado, y cada paso hacia atrás necesita los valores de su paso, así que todas las salidas intermedias deben guardarse hasta que la pasada hacia atrás llegue a ellas.

El paso hacia atrás de una capa

Prepare un pequeño experimento en una hoja de cálculo. Ponga una tabla de pesos W de 2 por 2 con filas (1, 2) y (3, 4), una entrada x igual a (1, 2) y un vector v igual a (1, -1) que pondera las dos salidas.

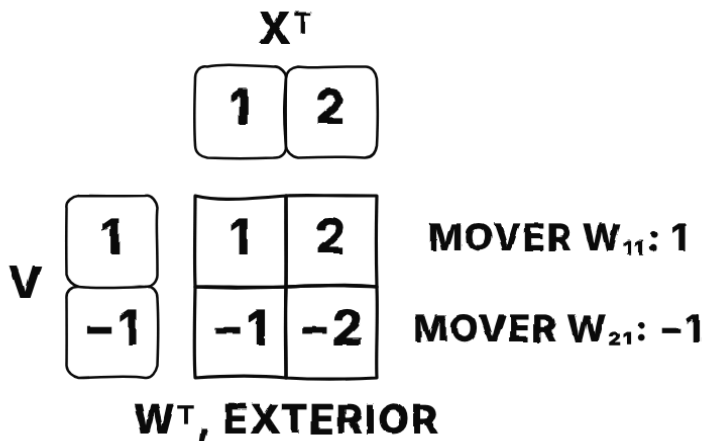
Calcule W por x , que es (5, 11), y después el número único v por eso, 1 por 5 menos 1 por 11, que es -6. Ahora mueva un poco el peso de arriba a la izquierda de 1 a 1,01 y mire el resultado: se mueve 0,01, de -6 a -5,99. Si mueve en cambio el peso de abajo a la izquierda, el resultado se mueve -0,01; aquí hay un patrón, y el libro lo convierte en una fórmula.



Tome la parte más sencilla de una capa totalmente conectada, una proyección lineal sin sesgo: $f(\mathbf{x}, \mathbf{W}) = \mathbf{W}\mathbf{x}$. Su jacobiano de entrada es simplemente la matriz $\mathbf{W}$, pero su jacobiano de los pesos es un tensor de rango 3: una dirección para las salidas y dos para las filas y columnas de $\mathbf{W}$. Es justo el tipo de tabla que el libro quiere evitar construir. Los VJP la evitan.

El VJP de entrada no tiene ninguna estructura especial: $\text{vjp}_{f,\mathbf{x}}(\mathbf{v}) = \mathbf{v}^T \mathbf{W}^T = [\mathbf{W}\mathbf{v}]^T$. Dicho con palabras: el adjunto que vuelve se multiplica por la traspuesta de los pesos. El libro escribe este producto con los pesos traspuestos, y el resultado vuelve a ser un vector. En nuestra hoja de cálculo, $\mathbf{W}$ por $(1, -1)$ es $(1 - 2, 3 - 4)$, es decir, $(-1, -1)$, el vector que la fórmula del libro devuelve hacia la entrada.

El VJP de los pesos resulta ser un producto exterior, sin ningún tensor de rango 3: $\text{vjp}_{f,\mathbf{w}}(\mathbf{v}) = \mathbf{v}\mathbf{x}^T$. El producto exterior de dos vectores es la tabla cuyo valor en la fila i y la columna j es el i -ésimo valor del primero por el j -ésimo valor del segundo. El libro lo deduce en un breve recuadro: el número $y = \mathbf{v}^T \mathbf{W} \mathbf{x} = \sum_i \sum_j W_{ij} v_i x_j$ contiene cada peso exactamente una vez, multiplicado por v_i y x_j . Así que $\partial y / \partial W_{ij} = v_i x_j$, que es la definición del producto exterior, valor a valor. Con nuestros números, $\mathbf{v}$ por $\mathbf{x}$ traspuesto tiene filas $(1, 2)$ y $(-1, -2)$: el valor de arriba a la izquierda, 1, es el 0,01 que vimos al mover el peso de arriba a la izquierda, dividido entre 0,01, y el de abajo a la izquierda, -1, coincide con el segundo movimiento.



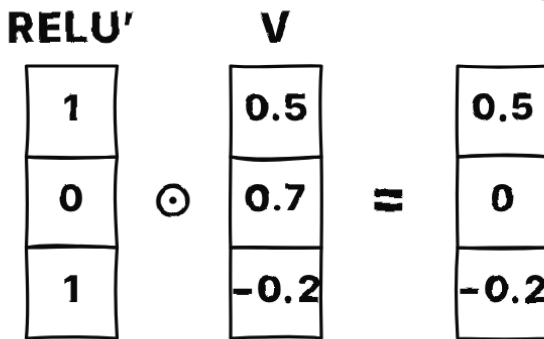
El VJP de los pesos es un producto exterior

La regla que hay que recordar es corta. Cada vez que la pasada hacia delante aplica una proyección lineal, la pasada hacia atrás multiplica el gradiente que viaja por la traspuesta de sus pesos y calcula el gradiente de $\mathbf{W}$ como un producto exterior. Cuando se entrena un asistente de chat, la mayoría de sus capas son proyecciones de este tipo, y estas dos operaciones son buena parte de lo que hace su pasada hacia atrás.

Ahora una función de activación elemento a elemento sin parámetros entrenables, como la ReLU: $f(\mathbf{x}, \emptyset) = \varphi(\mathbf{x})$. Sin pesos, solo hay un VJP de entrada. Su jacobiano es una matriz diagonal, cero en todas partes salvo en la diagonal, donde están las derivadas de φ : $[\partial_{\mathbf{x}}\varphi(\mathbf{x})]_{ii} = \varphi'(x_i)$, porque cada salida depende solo de su propia entrada. Una

matriz diagonal por un vector es solo un escalado, valor a valor, que se llama producto de Hadamard: $\text{vjp_x}(f, \mathbf{v}) = \mathbf{v} \odot \phi'(\mathbf{x})$. Para la ReLU, con $\mathbf{x}$ igual a $(2, -1, 3)$, las derivadas son $(1, 0, 1)$; un vector viajero $(0,5, 0,7, -0,2)$ sale como $(0,5, 0, -0,2)$. También aquí la matriz diagonal completa nunca se construye.

$$\mathbf{X} = (2, -1, 3)$$



The diagram illustrates the Hadamard product operation between two vectors, ReLU' and $\mathbf{V}$, resulting in a third vector. The ReLU' vector has values 1, 0, and 1. The $\mathbf{V}$ vector has values 0.5, 0.7, and -0.2. The resulting vector has values 0.5, 0, and -0.2. The operation is represented by the symbol $\odot$ and an equals sign.

ReLU'
1
0
1

 $\odot$

$\mathbf{V}$
0.5
0.7
-0.2

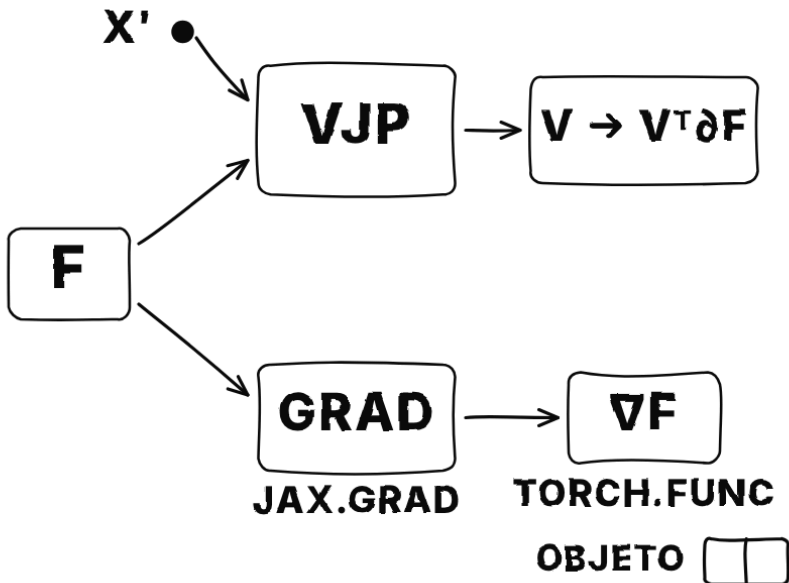
 $=$

0.5
0
-0.2

El paso atrás de la ReLU es una máscara

¿Cómo organizan todo esto los marcos de trabajo reales? Hay muchas maneras, dice el libro, desde las listas de Wengert, una lista grabada de las operaciones realizadas, como en TensorFlow, hasta transformaciones de código a código que reescriben el propio código del programa. Escribir las primitivas como funciones de la entrada y los pesos encaja con los marcos funcionales como JAX, donde todo es una función. Allí un VJP es una función de orden superior, una

función que recibe y devuelve funciones, con la firma $(\mathbb{R}^c \rightarrow \mathbb{R}^{c'}) \rightarrow \mathbb{R}^c \rightarrow (\mathbb{R}^{c'} \rightarrow \mathbb{R}^c)$. Dicho con palabras: dele una función f y un punto de entrada, y le devuelve una función nueva, que recibe un vector $\mathbf{v}$ del tamaño de la salida y devuelve $\mathbf{v}^T \partial f(\mathbf{x}')$.



Un VJP devuelve una función

El gradiente de una función con una sola salida es otra función de orden superior, con la firma $(\mathbb{R}^c \rightarrow \mathbb{R}) \rightarrow (\mathbb{R}^c \rightarrow \mathbb{R}^c)$: recibe f y devuelve una función que calcula $\nabla f(\mathbf{x})$. JAX ofrece funciones así, el libro nombra `jax.grad` entre ellas, y PyTorch copia la interfaz en su módulo `torch.func`. El ejemplo del libro toma la suma de los cuadrados de un vector, pide su función gradiente, la aplica a diez números aleatorios y recibe diez números, como debe ser: uno por entrada, aquí el doble de cada entrada. Una función así también puede trazarse, por ejemplo

con `torch.compile`, para obtener un grafo computacional optimizado, y combinar VJP y JVP da aún más, como las segundas derivadas. En la práctica, sin embargo, los modelos se construyen como objetos que guardan dentro sus parámetros, y el libro pasa a continuación a tender un puente entre los dos estilos.

Para recordar

Para una proyección lineal $\mathbf{W}\mathbf{x}$, el jacobiano de los pesos sería un tensor de rango 3, pero los VJP lo evitan: el VJP de entrada multiplica por la traspuesta de $\mathbf{W}$, y el VJP de los pesos es el producto exterior $\mathbf{v}\mathbf{x}^T$. El razonamiento del recuadro es sencillo: en $\mathbf{v}^T\mathbf{W}\mathbf{x}$ cada peso W_{ij} aparece una vez, multiplicado por v_i y x_j . Para una activación elemento a elemento el jacobiano es diagonal, así que su VJP es solo $\mathbf{v} \odot \phi'(\mathbf{x})$, un escalado, de nuevo sin construir la matriz. En marcos funcionales como JAX, un VJP y un gradiente son funciones de orden superior que reciben una función y devuelven otra nueva; PyTorch ofrece lo mismo en `torch.func`.

Preguntas

1. ¿Por qué el VJP de los pesos de una proyección lineal $\mathbf{W}\mathbf{x}$ es mucho más fácil de manejar que su jacobiano de los pesos?
 - a. Porque el jacobiano de los pesos es un tensor de rango 3, mientras que el VJP de los pesos es solo el producto exterior $\mathbf{v}\mathbf{x}^T$

- b. Porque el VJP de los pesos ignora la entrada $\mathbf{x}$
 - c. Porque el jacobiano de los pesos es simplemente $\mathbf{W}$
 - d. Porque el VJP de los pesos es siempre una matriz diagonal
2. ¿Por qué la derivada de $y = \mathbf{v}^T \mathbf{W} \mathbf{x}$ respecto de W_{ij} resulta ser $v_i x_j$?
- a. Porque $\mathbf{W}$ es una matriz cuadrada
 - b. Porque en la doble suma cada peso W_{ij} aparece exactamente una vez, multiplicado por v_i y x_j
 - c. Porque el jacobiano de entrada es la traspuesta de $\mathbf{W}$
 - d. Porque la ReLU pone a cero los demás valores
3. Para una activación elemento a elemento como la ReLU, ¿qué hace el VJP de entrada con el vector viajero $\mathbf{v}$?
- a. Multiplica $\mathbf{v}$ por la traspuesta de una matriz de pesos
 - b. Hace el producto exterior de $\mathbf{v}$ con la entrada
 - c. Escala cada valor de $\mathbf{v}$ por la derivada de ϕ en la entrada correspondiente, el producto de Hadamard $\mathbf{v} \odot \phi'(\mathbf{x})$
 - d. Deja $\mathbf{v}$ sin cambios, porque no hay pesos
4. En un marco funcional como JAX, ¿qué devuelve el VJP de una función?
- a. Un número: la derivada en un punto
 - b. Una función nueva que recibe un vector del tamaño de la salida y devuelve $\mathbf{v}^T \partial f(\mathbf{x}')$
 - c. La matriz jacobiana completa de la función
 - d. Una lista de las operaciones que realizó el programa

Respuestas

1. **a)** El jacobiano de los pesos tiene una dirección para las salidas y dos para las filas y columnas de $\mathbf{W}$; el VJP de los pesos lo reduce a una tabla con valores $v_i x_j$, con la forma de la propia $\mathbf{W}$.
2. **b)** Escrito por extenso, y es la suma de $W_{ij} v_i x_j$ para todos los i y j , así que mover un poco un peso cambia y en $v_i x_j$ veces ese movimiento.
3. **c)** El jacobiano es diagonal con valores $\phi'(x_i)$, y una matriz diagonal por un vector es un escalado valor a valor, así que nunca hay que construir la matriz.
4. **b)** El VJP es una función de orden superior: dada f y un punto de entrada, devuelve una función que convierte un vector del tamaño de la salida en $\mathbf{v}^T \partial f(\mathbf{x}')$.

Ejemplos resueltos

El gradiente de los pesos como producto exterior Siguiendo el recuadro del libro, tome $y = \mathbf{v}^T \mathbf{W} \mathbf{x}$ con $\mathbf{v} = (1, -1)$ y $\mathbf{x} = (1, 2)$. Escriba el gradiente de y respecto de la matriz $\mathbf{W}$ de 2 por 2.

Escriba y como la doble suma de $W_{ij} v_i x_j$: cada peso aparece una vez.

Así que $\partial y / \partial W_{ij} = v_i x_j$, que es el producto exterior $\mathbf{v} \mathbf{x}^T$.

Primera fila, $v_1 = 1$: $(1 \times 1, 1 \times 2) = (1, 2)$.

Segunda fila, $v_2 = -1$: $(-1 \times 1, -1 \times 2) = (-1, -2)$.

Respuesta: Una tabla de 2 por 2 con filas $(1, 2)$ y $(-1, -2)$.

Ejercicios

1. Calcule el VJP de los pesos $\mathbf{v} \mathbf{x}^T$ para $\mathbf{v} = (2, 1)$ y $\mathbf{x} = (3, 0, 1)$.

Pista: La fila i es v_i por todo $\mathbf{x}$.

Respuesta: Filas $(6, 0, 2)$ y $(3, 0, 1)$: la primera es 2 por $\mathbf{x}$, la segunda 1 por $\mathbf{x}$.

2. Una ReLU recibe $\mathbf{x} = (-2, 0, 5, 4)$, y la pasada hacia atrás trae el vector $\mathbf{v} = (3, 3, 3)$. ¿Qué devuelve su VJP de entrada?

Pista: La derivada de la ReLU es 0 para entradas negativas y 1 para las positivas; escale valor a valor.

Respuesta: $(0, 3, 3)$: las derivadas son $(0, 1, 1)$, y $\mathbf{v} \odot \phi'(\mathbf{x})$ multiplica valor a valor.

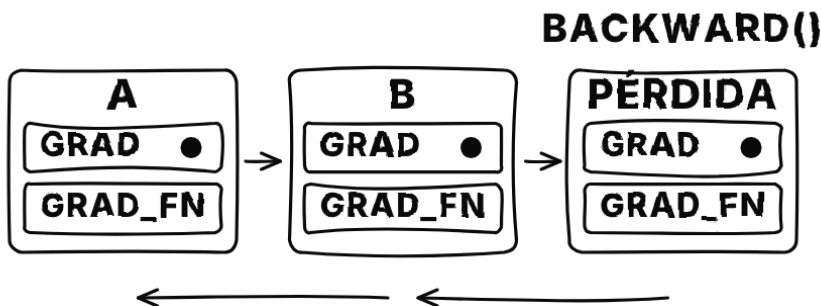
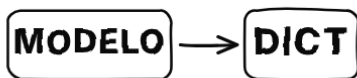
3. Explique por qué el jacobiano de una función de activación elemento a elemento es diagonal.

Pista: Pregúntese de qué entradas depende cada salida.

Respuesta: Cada salida $\phi(x_i)$ depende solo de su propia entrada x_i , así que mover cualquier otra entrada no la cambia: todos los valores fuera de la diagonal son cero, y en la diagonal está $\phi'(x_i)$.

Los marcos de trabajo, y por qué gana la ReLU

Antes de los marcos de trabajo, un pequeño puente. En PyTorch un modelo es un objeto que guarda dentro sus parámetros, mientras que el estilo funcional quiere una función de la entrada y los pesos. El libro muestra cómo «purificar» el objeto: reunir los parámetros con nombre del modelo en un diccionario y luego llamar al cálculo hacia delante del modelo mediante `torch.func.functional_call`, pasando explícitamente el modelo, ese diccionario y la entrada. Ahora los parámetros vuelven a ser un argumento, como en la notación del libro.



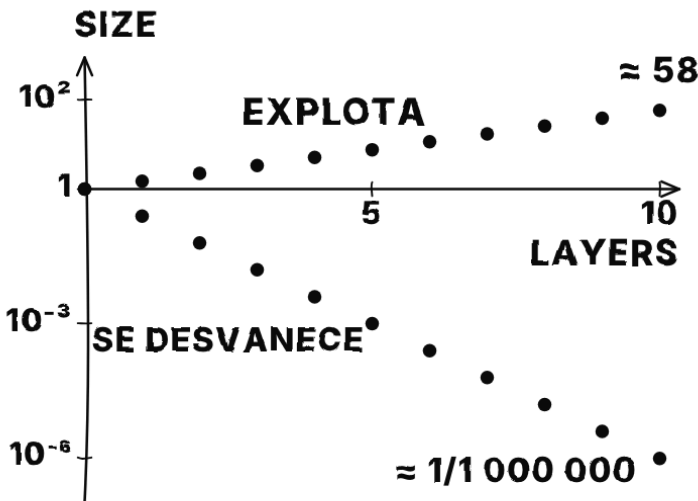
Un tensor recuerda quién lo hizo

De forma más general, PyTorch maneja directamente el estilo de objetos, sin esos pasos intermedios. La figura del libro dibuja un tensor, la tabla de PyTorch, como una caja con campos extra añadidos. A la izquierda, al crearse, el tensor lleva un hueco para su gradiente, vacío al principio, y una nota de la operación que lo produjo, la propiedad `grad_fn`. A la derecha, durante una pasada hacia atrás, esas notas se siguen en sentido inverso por el grafo computacional, y cada gradiente se guarda dentro del campo `grad` de su tensor, pero solo donde `requires_grad` se ha puesto explícitamente a verdadero, para no gastar memoria. Todo esto empieza cuando usted llama a `backward()` sobre un solo número, la pérdida; es el momento, en el código de entrenamiento de cualquier aplicación de fotos, en que aparecen los gradientes de todos los pesos. El autor

insiste en que esto es solo un panorama general, remite a la documentación oficial y sugiere con entusiasmo construir usted mismo un pequeño sistema en modo inverso, por ejemplo estudiando una implementación didáctica llamada micrograd.

Pruébelo

Tome una calculadora, escriba 1 y multiplique por 0,25 diez veces. Quizá espere algo pequeño pero utilizable. Lo que ve es 0,00000095, más o menos una millonésima. Ahora empiece otra vez desde 1 y multiplique en cambio por 1,5 diez veces: llega a unos 58, y cincuenta veces darían más de seiscientos millones.

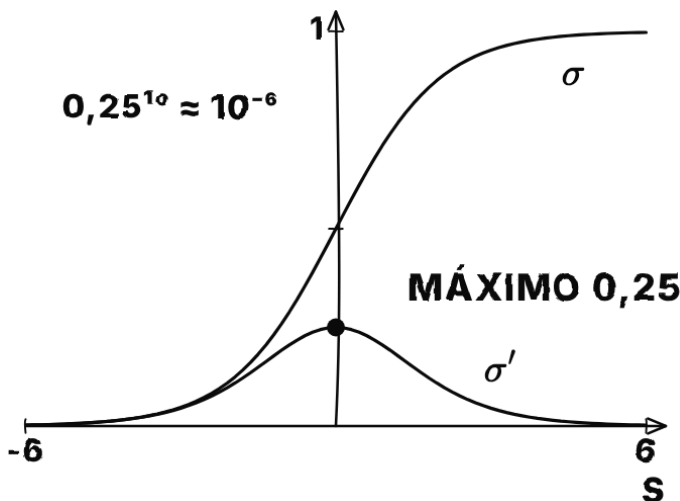


Diez capas de 0,25 y de 1,5

Ese juego de calculadora es la pasada hacia atrás de una red profunda. Por la parte de la ReLU sabemos que cada función de activación escala los adjuntos por $\phi'(x)$ en su camino de vuelta. Si $\phi'(\cdot) < 1$ en todas partes, el gradiente se reduce hacia cero exponencialmente rápido con el número de capas: el problema del desvanecimiento del gradiente. Si $\phi'(\cdot) > 1$ en todas partes, crece exponencialmente hacia el infinito: el problema de la explosión del gradiente. Son problemas serios en la práctica, porque las bibliotecas guardan los números con una precisión limitada, normalmente de 32 bits o menos, así que basta con unas pocas docenas de capas para que un número se hunda en el cero o se desborde.

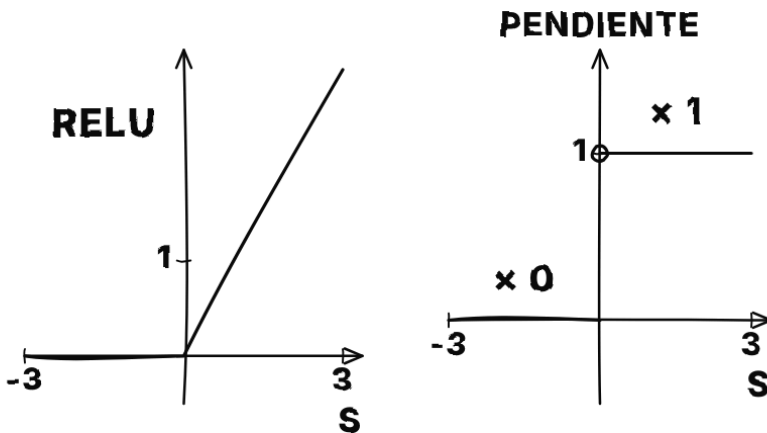
El libro añade un recuadro curioso. Una pila de capas puramente lineales, calculada en coma flotante, no es exactamente lineal, por diminutos saltos en los límites de la precisión de la máquina. En general no es un problema, pero el autor señala que se ha aprovechado para entrenar redes profundas hechas solo de capas lineales.

Ahora la sigmoide, la curva en forma de S, $\sigma(s)$, que fue popular en el pasado como versión suave de la función escalón. Su derivada es $\sigma'(s) = \sigma(s)(1 - \sigma(s))$. Dicho con palabras: la pendiente es el valor por lo que le falta para llegar a 1. Como σ está siempre entre 0 y 1, este producto es máximo cuando σ vale un medio, lo que da un medio por un medio: así que $\sigma'(s) \in [0, 0.25]$. En s igual a 0, σ vale 0,5 y la pendiente es exactamente 0,25; ese es el factor de nuestra calculadora, el mejor caso, y diez capas sigmoide ya reducen el gradiente un millón de veces. La gráfica del libro lo muestra: la sigmoide roja sube de 0 a 1, mientras que su derivada verde es una joroba baja con el máximo en un cuarto, que se desvanece hacia los dos lados.



La pendiente de la sigmoide tiene su máximo en un cuarto

¿Podría una función de activación evitar los dos problemas? Solo la identidad, que lo deja todo igual, tiene $\phi'(s) = 1$ en todas partes, y la identidad haría que las capas se fundieran en una sola aplicación lineal. Así que necesitamos una función lo bastante lineal para mantener sanos los gradientes, pero lo bastante no lineal para separar las capas. La ReLU encaja: $\partial_s \text{ReLU}(s) = 0$ para $s < 0$, 1 para $s > 0$. O el gradiente se anula, lo que hace disperso el cálculo, o se multiplica exactamente por 1 , sin ningún escalado; en la segunda gráfica del libro la ReLU roja es plana y luego sube en diagonal, y la derivada verde es un escalón de 0 a 1 .



SOBRESCRIBE

La pendiente de la ReLU es 0 o 1

Un regalo más de la ReLU. Su gradiente es el mismo tanto si se calcula a partir de la entrada de la capa como de su salida, porque la ReLU solo enmascara los valores negativos y deja intactos los positivos: una salida positiva significa una entrada positiva. Así que en modo inverso la entrada de la capa puede sobrescribirse con su salida durante la pasada hacia delante, ahorrando algo de memoria sin perjudicar la corrección de la diferenciación automática; PyTorch lo ofrece mediante un parámetro para trabajar in situ.

Para recordar

En PyTorch cada tensor recuerda la operación que lo hizo; llamar a backward sobre la pérdida sigue esas notas en sentido inverso y guarda los gradientes dentro de los tensores que los piden. Cada función de activación multiplica el gradiente de vuelta por su derivada, así que factores menores que 1 hacen que los gradientes se desvanezcan y factores mayores que 1 hacen que exploten, exponencialmente con el número de capas, y la precisión limitada hace que esto se note pronto. La derivada de la sigmoide, $\sigma(s)(1 - \sigma(s))$, es como mucho 0,25, una de las causas principales del desvanecimiento del gradiente. La derivada de la ReLU es 0 o exactamente 1, así que ni encoge ni infla lo que pasa, e incluso permite que la pasada hacia delante sobrescriba su entrada para ahorrar memoria.

Preguntas

1. En PyTorch, ¿qué permite que una llamada a backward() encuentre el camino de vuelta por el cálculo?
 - a. Cada tensor lleva una nota de la operación que lo creó, y esas notas se siguen en sentido inverso
 - b. PyTorch vuelve a ejecutar todo el programa de forma simbólica
 - c. Los gradientes se calculan numéricamente, con dos llamadas por peso

- d. Cada tensor guarda el jacobiano completo del paso que lo hizo
2. Una red profunda usa una activación cuya derivada es menor que 1 en todas partes. ¿Qué les pasa a los gradientes que llegan a las primeras capas?
- a. Crecen exponencialmente con el número de capas
 - b. Se mantienen iguales, porque los pesos lo compensan
 - c. Encogen hacia cero exponencialmente rápido con el número de capas
 - d. Pasan a valer exactamente 1
3. ¿Por qué la sigmoide es una candidata de primera al desvanecimiento del gradiente?
- a. Porque su derivada puede ser mayor que 1
 - b. Porque su derivada $\sigma(s)(1 - \sigma(s))$ nunca pasa de 0,25
 - c. Porque no es diferenciable en 0
 - d. Porque es una función lineal
4. ¿Por qué puede la ReLU sobrescribir su entrada con su salida en la pasada hacia delante sin estropear el modo inverso?
- a. Porque la ReLU no tiene ninguna derivada que calcular
 - b. Porque su gradiente se lee igual de bien a partir de la salida: las salidas positivas vienen de entradas positivas, y las negativas se enmascaran
 - c. Porque PyTorch guarda de todos modos una segunda copia
 - d. Porque la pasada hacia atrás nunca visita las capas de activación

Respuestas

1. **a)** Los tensores llevan añadida la operación que los generó (`grad_fn`); `backward()` la sigue en sentido inverso y guarda los gradientes dentro de los tensores que los requieren.
2. **c)** Cada activación escala los adjuntos por ϕ' en el camino de vuelta, así que factores menores que 1 se acumulan hasta dar un gradiente que se desvanece.
3. **b)** Con σ entre 0 y 1, el producto $\sigma(1 - \sigma)$ es como mucho un medio por un medio, así que cada capa sigmoide reduce el gradiente al menos a la cuarta parte.
4. **b)** La ReLU solo enmascara los valores negativos y deja intactos los positivos, así que su gradiente es el mismo tanto si se calcula a partir de la entrada como de la salida.

Ejemplos resueltos

La pendiente máxima de la sigmoide Con $\sigma'(s) = \sigma(s)(1 - \sigma(s))$ y sabiendo que $\sigma(s)$ está entre 0 y 1, halle el valor máximo posible de σ' y el factor por el que diez capas sigmoide reducen un gradiente, en el mejor caso.

Escriba la derivada como el producto de dos números que suman 1: σ y $1 - \sigma$.

Un producto así es máximo cuando los dos son iguales, en $\sigma = 0,5$, lo que da $0,5 \times 0,5 = 0,25$.

Así que $\sigma'(s)$ está entre 0 y 0,25, y cada capa sigmoide multiplica el gradiente de vuelta como mucho por 0,25.

En diez capas: 0,25 elevado a diez, unos 0,00000095.

Respuesta: Como mucho 0,25 por capa; diez capas reducen el gradiente, en el mejor caso, a más o menos una millonésima.

Ejercicios

1. En cierta entrada el valor de la sigmoide es $\sigma = 0,8$. ¿Cuál es allí su pendiente?

Pista: Multiplique el valor por lo que le falta para llegar a 1.

Respuesta: $0,8 \times 0,2 = 0,16$, por debajo del máximo de 0,25.

2. Suponga que cada capa de una red de diez capas multiplicara el gradiente de vuelta por 2. ¿Por qué factor quedaría escalado el gradiente en la primera capa? ¿Cómo se llama este problema?

Pista: Eleve el factor al número de capas.

Respuesta: 2 elevado a diez, 1024 : el gradiente explota, que es el problema de la explosión del gradiente.

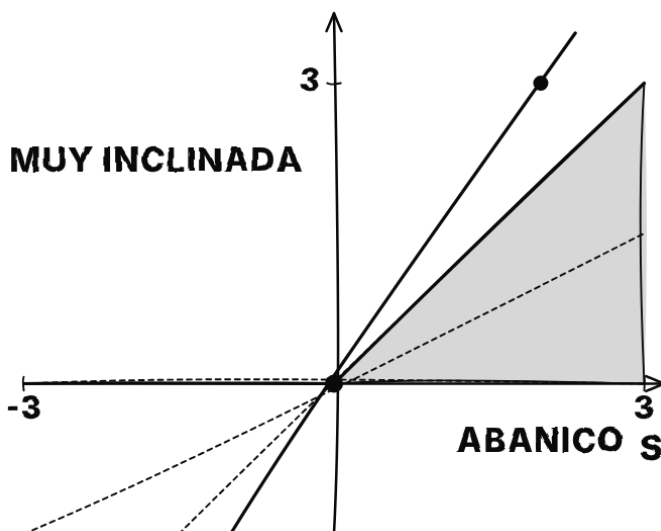
3. Explique por qué la ReLU evita tanto el desvanecimiento como la explosión del gradiente en las entradas donde está activa.

Pista: Fíjese en los dos valores que puede tomar su derivada.

Respuesta: Su derivada es 0 para las entradas negativas y exactamente 1 para las positivas. Donde está activa, el gradiente se multiplica exactamente por 1, así que ni encoge ni se infla; en el resto se pone a cero, lo que solo hace disperso el cálculo.

La esquina de la ReLU

Dibuje la ReLU en una hoja de papel: plana sobre el cero para los números negativos, y luego subiendo por la diagonal para los positivos, con una esquina afilada en el origen. Ahora ponga una regla que pase por esa esquina. Inclínela en horizontal, y toda la gráfica queda sobre ella o por encima; levántela hasta la diagonal, y la gráfica sigue quedando sobre ella o por encima. Cualquier inclinación intermedia también sirve, pero inclínela más que la diagonal y la gráfica queda por debajo de ella a la derecha. En la esquina no hay una sola pendiente, sino todo un abanico, y ese abanico es la forma en que el libro resuelve una cuestión que había dejado abierta hasta ahora.

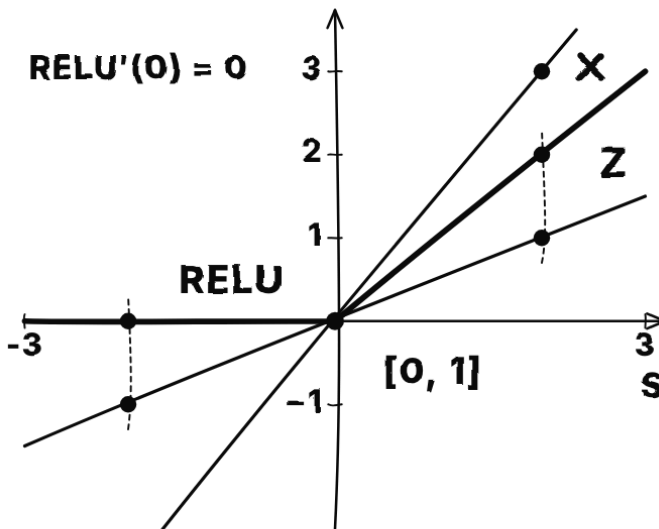


Un abanico de reglas en la esquina

La ReLU no tiene derivada en 0, así que una red construida con ella no es suave. ¿Y entonces? El libro da primero la respuesta pragmática: al entrenar con descenso de gradiente estocástico desde un punto de partida aleatorio y distinto de cero, la probabilidad de caer exactamente en s igual a 0 es prácticamente nula, mientras que el gradiente existe en $\text{ReLU}(\epsilon)$ para cualquier ϵ distinto de 0, por pequeño que sea. En el entrenamiento de un asistente de chat, con miles de millones de números circulando, un cero exacto es un accidente raro.

La respuesta técnica es el subgradiente. Para una función convexa f , una cuya gráfica se curva hacia arriba en todas partes, un subgradiente en x es un número z tal que, para todo y , $f(y) \geq f(x) + z(y - x)$.

– x). Dicho con palabras: la recta que pasa por el punto $(x, f(x))$ con pendiente z queda sobre toda la gráfica o por debajo de ella, como nuestra regla. Recuerda a la definición de convexidad, y donde f tiene derivada solo existe una recta así, la tangente, de modo que el subgradiente es simplemente la derivada. En una esquina sirven muchas pendientes, y juntas forman el subdiferencial: $\partial_x f(x)$ es el conjunto de los z que son subgradiientes de $f(x)$.

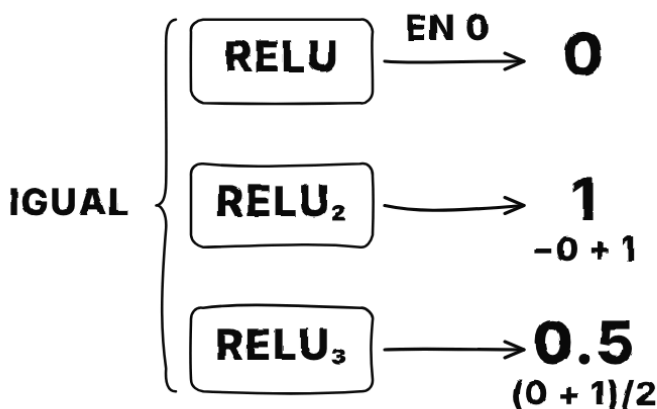


Comprobar las pendientes 0,5 y 1,5

Ahora puede completarse la ReLU sustituyendo su gradiente por su subdiferencial en 0: $\partial_s \text{ReLU}(s) = 0$ para $s < 0$, 1 para $s > 0$, y todo $[0, 1]$ para $s = 0$. Compruebe un valor con números: tome z igual a 0,5 en la esquina. En y igual a 2 la recta da 1 y la ReLU da 2; en y igual a -2 la recta da -1 y la ReLU da 0; la recta queda por debajo, como debe.

Tome z igual a 1,5 y en y igual a 2 la recta da 3, por encima del 2 de la ReLU: no es un subgradiente, igual que mostraba la regla demasiado inclinada. Así que cualquier valor de 0 a 1 es válido en la esquina, y la mayoría de las implementaciones eligen $\text{ReLU}'(0) = 0$; elegir un subgradiente en cada paso del descenso se llama descenso por subgradiente.

La historia completa es más delicada. Para funciones que no son convexas, puede no existir ningún subgradiente, y se recurre a generalizaciones que relajan la definición a un pequeño entorno de x , como el subdiferencial de Clarke. La diferenciación automática añade su propia trampa: distintas implementaciones de la misma función pueden devolver subgradientes distintos, algunos ni siquiera válidos, y una demostración formal necesita reglas de la cadena más refinadas. La nota del libro da un ejemplo. Defina $\text{ReLU}_2(s) = \text{ReLU}(-s) + s$ y $\text{ReLU}_3(s) = 0,5(\text{ReLU}(s) + \text{ReLU}_2(s))$. Pruebe algunos números y los dos dan exactamente la ReLU, y sin embargo una pasada hacia atrás en PyTorch en 0 devuelve 0 para la ReLU, 1 para ReLU_2 y 0,5 para ReLU_3 . La razón es la regla de la cadena en acción: la derivada de ReLU_2 es menos $\text{ReLU}'(0)$, que es 0, más 1, y ReLU_3 promedia 0 y 1.



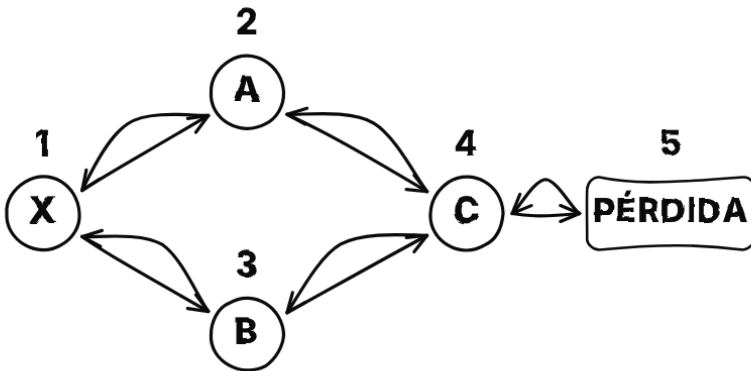
La misma función, tres respuestas

A partir de aquí, el libro pasa a la práctica. Si hizo los ejercicios del capítulo anterior, ya ha usado el modo inverso en PyTorch y JAX, y este capítulo explica lo que ocurría por dentro. El autor sugiere reconstruir usted mismo un pequeño sistema en modo inverso, para valores escalares, con micrograd como buen modelo didáctico. Un detalle que el capítulo se saltó: en un grafo general sin bucles, las variables deben ordenarse antes de la pasada hacia atrás, o los gradientes circulan por caminos equivocados; micrograd lo hace con una ordenación topológica barata, un orden en el que cada valor va después de todo aquello de lo que depende. Otro ejercicio es escribir una primitiva nueva en PyTorch dando su pasada hacia delante y sus VJP, por ejemplo una función de activación entrenable del capítulo anterior, aunque PyTorch también podría deducir esa pasada hacia

atrás por sí mismo. En JAX existen los mismos caminos: una versión didáctica de JAX llamada autodidax, escribir una primitiva con su propio VJP, y el JAX Autodiff Cookbook para derivadas superiores, hessianas y más.

MICROGRAD

AUTODIDAX



Ordenar el grafo y luego volver

Así se cierra el capítulo: de una cadena en una hoja de cálculo al motor que entrena cada modelo de su teléfono, todo construido sobre una idea, multiplicar hacia atrás.

Para recordar

La ReLU no tiene derivada en 0, pero el entrenamiento casi nunca cae exactamente ahí, y el gradiente existe en todos los demás puntos. Un subgradiente en x es una pendiente z cuya recta por $(x, f(x))$ queda por debajo de toda la gráfica; en un punto suave es la derivada, en una esquina hay todo un conjunto, el subdiferencial. Para la ReLU en 0 es válido cualquier valor de 0 a 1, las implementaciones suelen tomar 0, y fórmulas distintas pero equivalentes pueden dar respuestas distintas en un marco de trabajo, como muestran ReLU₂ y ReLU₃. El libro termina invitándole a construir usted mismo el modo inverso, con micrograd o autodidax, y a escribir primitivas con sus propios VJP.

Preguntas

1. ¿Cuál es la respuesta pragmática del libro a que la ReLU no tenga derivada en 0?
 - a. El entrenamiento parte de un punto aleatorio distinto de cero, así que caer exactamente en 0 es prácticamente imposible, y el gradiente existe en todos los demás puntos
 - b. Cerca de 0, la ReLU se sustituye por la sigmoide
 - c. La red se suaviza antes de entrenarla
 - d. Los marcos de trabajo se niegan a derivar en 0 y detienen el entrenamiento

2. En un punto donde una función convexa tiene derivada, ¿cuántos subgradiientes tiene?
- a. Ninguno
 - b. Exactamente uno, la propia derivada
 - c. Todo un intervalo
 - d. Dos, uno por cada lado
3. ¿Por qué una pendiente de 1,5 no es un subgradiente de la ReLU en 0?
- a. Porque los subgradiientes deben ser negativos
 - b. Porque la recta de pendiente 1,5 por la esquina sube por encima de la ReLU para entradas positivas, por ejemplo 3 frente a 2 en $y = 2$
 - c. Porque en la esquina solo 0 es un subgradiente válido
 - d. Porque la ReLU no es convexa
4. $\text{ReLU}_2(s) = \text{ReLU}(-s) + s$ es igual a la ReLU en todas partes. ¿Por qué la pasada hacia atrás de PyTorch en 0 devuelve 1 para ella, pero 0 para la ReLU?
- a. Porque ReLU_2 es una función distinta de la ReLU
 - b. Porque PyTorch usa diferenciación numérica para ReLU_2
 - c. Porque la regla de la cadena combina la elección de la implementación, $\text{ReLU}'(0) = 0$, con la derivada de la s sumada, lo que da $0 + 1$
 - d. Porque ReLU_2 no es continua en 0

Respuestas

1. a) Con descenso de gradiente estocástico desde una inicialización aleatoria distinta de cero, la probabilidad de dar exactamente con $s = 0$ es

prácticamente nula, mientras que el gradiente está definido en cualquier entrada distinta de cero.

2. b) Donde f es diferenciable, solo una recta que pase por el punto queda por debajo de toda la gráfica, la tangente, así que el único subgradiente es la derivada.

3. b) La recta de un subgradiente debe quedar sobre toda la gráfica o por debajo de ella; en $y = 2$ la recta inclinada da 3 mientras que la ReLU da 2.

4. c) La pasada hacia atrás sigue la fórmula tal como está escrita: menos $\text{ReLU}'(o)$, que se toma como 0, más 1 de la s sumada, así que fórmulas equivalentes pueden devolver subgradientes distintos.

Ejemplos resueltos

Las ReLU₂ y ReLU₃ del libro en cero Con la elección de la implementación $\text{ReLU}'(o) = 0$, siga la regla de la cadena para hallar qué devuelve una pasada hacia atrás en $s = 0$ para $\text{ReLU}_2(s) = \text{ReLU}(-s) + s$ y $\text{ReLU}_3(s) = 0,5(\text{ReLU}(s) + \text{ReLU}_2(s))$.

Para ReLU₂, el primer término, $\text{ReLU}(-s)$, da menos $\text{ReLU}'(o)$, es decir 0, por el signo menos interior.

El segundo término, s , da 1, así que ReLU₂ devuelve $0 + 1 = 1$.

Para ReLU₃, la pasada hacia atrás promedia las dos piezas: 0,5 por $(0 + 1)$.

Eso da 0,5, mientras que la ReLU sin más da 0.

Respuesta: 1 para ReLU₂ y 0,5 para ReLU₃, frente a 0 para la ReLU.

Ejercicios

1. Compruebe si $z = 0,3$ es un subgradiente de la ReLU en 0, probando $y = 4$ e $y = -4$ en $f(y) \geq f(o) + z(y - o)$.

Pista: Calcule el valor de la recta, $0,3y$, y compárelo con $\text{ReLU}(y)$.

Respuesta: En $y = 4$ la recta da 1,2 y la ReLU 4; en $y = -4$ la recta da -1,2 y la ReLU 0. La recta queda por debajo en los dos casos, y de hecho todo z entre 0 y 1 es un subgradiente en 0.

2. Muestre que $\text{ReLU}_2(s) = \text{ReLU}(-s) + s$ es igual a la ReLU, probando $s = 3$ y $s = -3$.

Pista: Calcule primero $\text{ReLU}(-s)$ y después sume s .

Respuesta: Para $s = 3$: $\text{ReLU}(-3) = 0$, más 3 da $3 = \text{ReLU}(3)$. Para $s = -3$: $\text{ReLU}(3) = 3$, más -3 da $0 = \text{ReLU}(-3)$.

3. Explique por qué un sistema en modo inverso sobre un grafo general sin bucles debe ordenar sus variables antes de la pasada hacia atrás.

Pista: Piense en un valor que alimenta a dos pasos posteriores.

Respuesta: El adjunto de un valor solo está completo cuando todos los pasos que lo usan han devuelto su contribución. Procesar las variables en un orden topológico, cada una después de todo aquello de lo que depende, y recorrer ese orden al revés, garantiza que ningún gradiente se pase antes de estar completo; si no, los gradientes seguirían caminos equivocados. micrograd lo hace con una ordenación topológica barata.