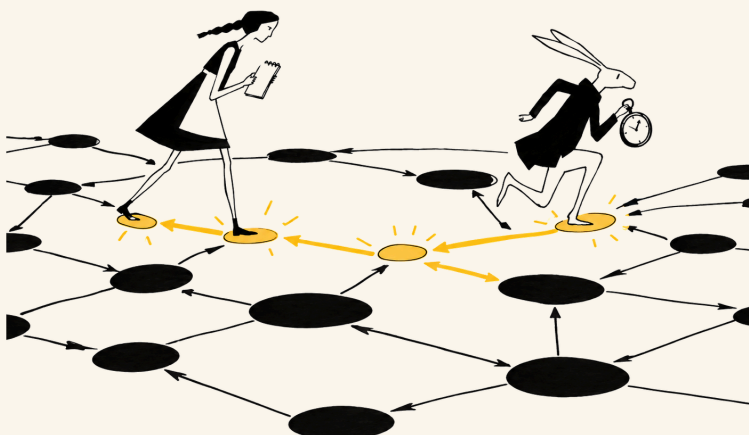


Las aventuras de Alicia en el País de las Maravillas Diferenciables



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

Diferenciación automática

Enunciado del problema

Diferenciación en modo directo

Diferenciación en modo inverso

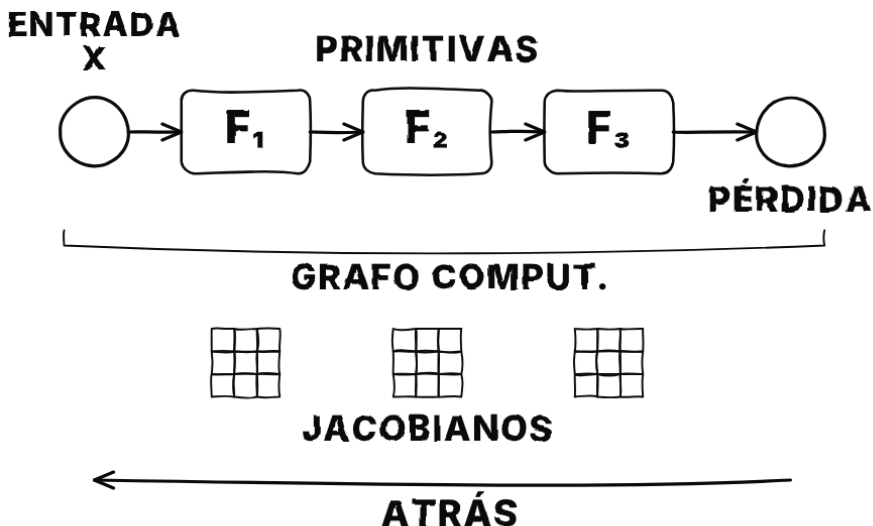
Productos vector-jacobiano

Implementar un sistema en modo inverso

Elegir una función de activación

Diferenciación automática

El capítulo anterior puso de relieve la necesidad de un procedimiento eficiente y automático para calcular los gradientes de cualquier secuencia posible de operaciones. En este capítulo describimos ese método, llamado retropropagación en la literatura de redes neuronales, o diferenciación automática en modo inverso en la de informática. Su análisis aporta varias ideas, que van desde la elección del modelo hasta la memoria que hace falta para optimizarlo.



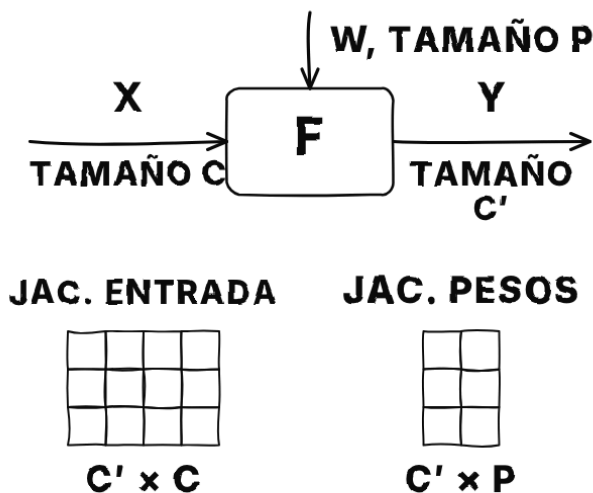
Un programa como cadena de primitivas

Planteamiento del problema.

Consideramos el problema de calcular de forma eficiente los gradientes de grafos computacionales genéricos, como los que surgen al optimizar una función de pérdida escalar en un modelo totalmente conectado, una tarea llamada diferenciación automática, o AD. Puede pensar en un grafo computacional como el conjunto de operaciones atómicas, que llamamos primitivas, que se obtiene al ejecutar el propio programa. Por brevedad consideraremos grafos secuenciales, pero todo se extiende con facilidad a grafos acíclicos e incluso a grafos computacionales más generales.

El problema puede parecer trivial, ya que la regla de la cadena de los jacobianos, de la sección sobre gradientes y jacobianos, nos dice que el gradiente de una composición de funciones es simplemente el producto matricial de las matrices jacobianas correspondientes. Sin embargo, implementar este proceso de forma eficiente es el reto central de este capítulo, y el algoritmo resultante, la AD en modo inverso o retropropagación, es una piedra angular de las redes neuronales y de la programación diferenciable en general. Entenderlo es también clave para entender el diseño, y las diferencias, de la mayoría de los entornos para implementar y entrenar estos programas, como TensorFlow, PyTorch o JAX.

Para plantear el problema, suponemos que tenemos a nuestra disposición un conjunto de primitivas, $y = f_i(\mathbf{x}, \mathbf{w}_i)$. Cada primitiva representa una operación sobre un vector de entrada $\mathbf{x} \sim (c_i)$, parametrizada por el vector $\mathbf{w}_i \sim (p_i)$, por ejemplo los pesos de una proyección lineal, y que da como salida otro vector $y \sim (c'_i)$.



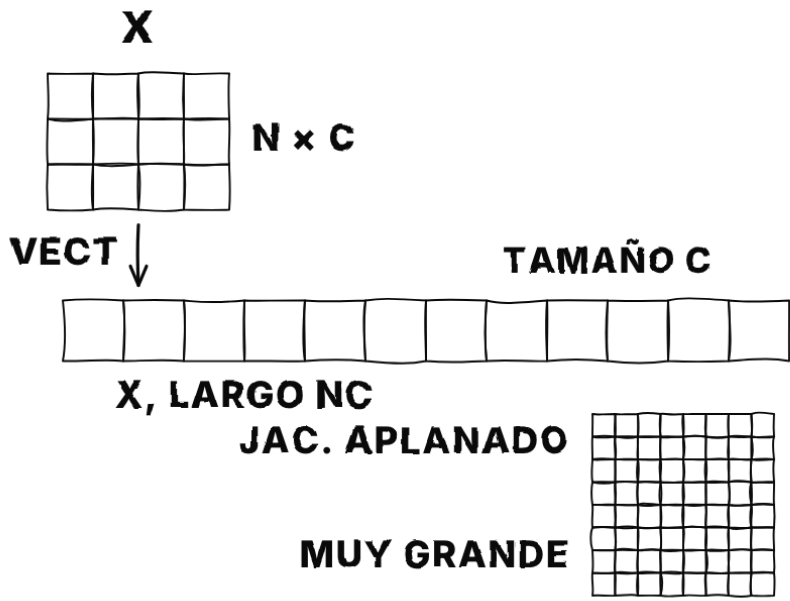
Una primitiva y sus dos jacobianos

Nuestra definición de primitiva es muy flexible: puede representar operaciones básicas de álgebra lineal como la multiplicación de matrices, capas en el sentido del capítulo 5, como una capa totalmente conectada con una función de activación, o incluso bloques o modelos más grandes. Esta componibilidad recursiva es una propiedad clave de la programación y se extiende a nuestro caso.

Solo suponemos que, para cada primitiva, sabemos calcular las derivadas parciales respecto de sus dos argumentos de entrada, que llamamos el jacobiano de entrada y el jacobiano de los pesos de la operación. El jacobiano de entrada es $\partial_x[f(\mathbf{x}, \mathbf{w})] \sim (c', c)$. El jacobiano de los pesos es $\partial_{\mathbf{w}}[f(\mathbf{x}, \mathbf{w})] \sim (c', p)$.

Son suposiciones razonables, ya que restringimos nuestro análisis a modelos diferenciables. Las primitivas continuas con uno o más puntos no diferenciables, como la ReLU, pueden encajar en este marco mediante subgradientes, como veremos al final de este capítulo. También pueden incluirse operaciones no diferenciables, como el muestreo o la umbralización, buscando una relajación de su gradiente o un estimador equivalente. Este último caso lo tratamos en el próximo volumen.

Sobre nuestra notación y los jacobianos de orden superior.



Aplanar un lote en un vector

Por legibilidad solo consideramos magnitudes vectoriales, ya que así todos los gradientes resultantes son matrices. En la práctica, las

primitivas existentes pueden tener entradas, pesos o salidas de rango mayor. Por ejemplo, considere una capa totalmente conectada básica sobre una entrada por minilotes, $f(\mathbf{X}, \mathbf{W}) = \mathbf{X}\mathbf{W} + \mathbf{b}$. En este caso la entrada $\mathbf{X}$ tiene forma (n, c) , los pesos tienen formas (c, c') y (c') , con c' un hiperparámetro, y la salida tiene forma (n, c') . Por tanto, el jacobiano de entrada tiene forma (n, c', n, c) y el jacobiano de los pesos tiene forma (n, c', c, c') , ambos de rango cuatro.

En nuestra notación podemos considerar los vectores aplanados equivalentes $\mathbf{x} = \text{vect}(\mathbf{X})$ y $\mathbf{w} = [\text{vect}(\mathbf{W}); \mathbf{b}]$, y nuestros jacobianos aplanados resultantes tienen formas (nc', nc) y (nc', cc') , respectivamente. Esto es crucial en lo que sigue, porque cada vez que hablemos de «el tamaño de entrada c » nos referimos a «el producto de todas las dimensiones de entrada», incluidas las del minilote. Esto muestra también que, aunque sepamos calcular los jacobianos, quizá no queramos materializarlos por completo en memoria, por su enorme dimensión.

Como nota final, nuestra notación coincide con la forma en que estas primitivas se implementan en una biblioteca funcional, como JAX. En un entorno orientado a objetos, como TensorFlow o PyTorch, vimos en el capítulo anterior que las capas se implementan como objetos, con los parámetros como propiedad del objeto y la llamada a la función sustituida por un método del objeto. Este estilo simplifica ciertas prácticas, como aplazar la inicialización de todos los parámetros hasta conocer las formas de entrada, la llamada inicialización perezosa, pero añade una pequeña capa de abstracción que hay que tener en cuenta al traducir nuestra notación a código que funcione. Como veremos, estas diferencias se reflejan a su vez en la forma en que se implementa la AD en los dos tipos de entornos.

Preguntas

1. ¿Por qué el capítulo trata el cálculo de gradientes como un problema real, si la regla de la cadena ya dice que el gradiente de una composición es un producto de jacobianos?
 - a. Porque la regla de la cadena no vale para matrices
 - b. Porque implementar ese producto de forma eficiente es la parte difícil
 - c. Porque los jacobianos de las primitivas son desconocidos
 - d. Porque la regla de la cadena solo se aplica a funciones escalares
2. ¿Qué supone el planteamiento que conocemos de cada primitiva?
 - a. Solo su salida para cada entrada posible
 - b. Sus derivadas parciales respecto de su entrada y respecto de sus pesos
 - c. El gradiente de la pérdida final respecto de sus pesos
 - d. Una fórmula simbólica para todo el programa
3. Una capa totalmente conectada trabaja con un minilote de entrada de forma (n, c) y salida de forma (n, c') . ¿Por qué podríamos no querer guardar su jacobiano de entrada en memoria?
 - a. Porque siempre es cero
 - b. Porque, aplanado, tiene forma (nc', nc) , que puede ser muy grande
 - c. Porque tiene rango 1 y no contiene información

- d. Porque los entornos orientados a objetos no pueden guardar matrices
4. ¿Cómo puede encajar en este marco una primitiva como la ReLU, que no es diferenciable en un punto?
- a. Dejándola fuera del grafo computacional
 - b. Sustituyéndola por una capa lineal
 - c. Mediante subgradientes
 - d. Tomando su salida al azar

Respuestas

1. **b)** La regla de la cadena da la respuesta como un producto matricial de jacobianos, pero implementar este proceso de forma eficiente es el reto central del capítulo.
2. **b)** Solo suponemos que, para cada primitiva, sabemos calcular el jacobiano de entrada y el jacobiano de los pesos, sus derivadas parciales respecto de sus dos argumentos.
3. **b)** Su jacobiano de entrada tiene rango cuatro y, aplanado, tiene forma (nc', nc) ; por esta enorme dimensión quizá no queramos materializarlo por completo en memoria.
4. **c)** Las primitivas continuas con puntos no diferenciables, como la ReLU, pueden encajar en este marco mediante subgradientes.

Ejemplos resueltos

Tamaño de un jacobiano aplanado Una capa totalmente conectada recibe un minilote de $n = 32$ entradas, cada una con $c = 10$ características, y produce $c' = 5$ salidas por entrada. ¿Cuántos números tiene su jacobiano de entrada aplanado, y cuántos su jacobiano de los pesos aplanado (sin contar el sesgo)?

El jacobiano de entrada aplanado tiene forma (nc', nc) , porque cada número de salida se deriva respecto de cada número de entrada.

Aquí $nc' = 32 \times 5 = 160$ y $nc = 32 \times 10 = 320$, así que la forma es $(160, 320)$.

Eso da $160 \times 320 = 51\,200$ números.

El jacobiano de los pesos aplanado tiene forma (nc', cc') , con $cc' = 10 \times 5 = 50$ pesos en W .

Eso da $160 \times 50 = 8000$ números, muchos más que los propios 50 pesos.

Respuesta: 51 200 números para el jacobiano de entrada y 8000 para el jacobiano de los pesos

Ejercicios

1. Una capa tiene entrada de forma (n, c) con $n = 2$ y $c = 3$, y salida de forma (n, c') con $c' = 4$. Escriba la forma de rango 4 de su jacobiano de entrada y su forma aplanada. ¿Tienen ambas el mismo número de casillas?

Pista: La forma de rango 4 es (n, c', n, c) ; la aplanada multiplica las dos primeras dimensiones y las dos últimas.

Respuesta: La forma de rango 4 es $(2, 4, 2, 3)$ y la forma aplanada es $(8, 6)$. Ambas tienen 48 casillas: aplanar solo reordena los números.

2. En este capítulo, «el tamaño de entrada c » de una primitiva significa el producto de todas sus dimensiones de entrada. ¿Por qué importa esto cuando un modelo se entrena con minilotes?

Pista: Piense en qué le ocurre al tamaño de entrada cuando se incluye la dimensión del minilote.

Respuesta: La dimensión del minilote forma parte del tamaño de entrada, así que c crece con el lote. Los jacobianos, cuyas formas se construyen a partir de estos tamaños, crecen con él, y por eso podemos saber calcularlos y aun así no querer guardarlos en memoria.

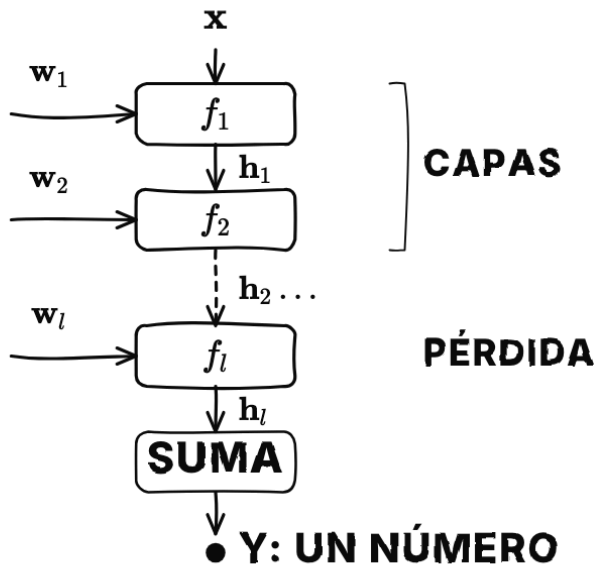
3. Explique con sus propias palabras la diferencia entre cómo trata los parámetros de una capa una biblioteca funcional como JAX y un entorno orientado a objetos como PyTorch, y qué relación tiene con la notación $f(x, w)$.

Pista: Pregúntese dónde viven los pesos en cada caso.

Respuesta: En la notación $f(x, w)$, los pesos son un segundo argumento explícito de la función, que es como los trata una biblioteca funcional como JAX. En un entorno orientado a objetos la capa es un objeto, los parámetros son una propiedad de ese objeto y la llamada a la función es un método; esto facilita cosas como la inicialización perezosa, pero añade una capa de abstracción entre la notación y el código.

Enunciado del problema

Resueltos todos estos detalles, estamos listos para enunciar la tarea de la AD. Considere una secuencia de l llamadas a primitivas, seguida de una suma final. Primero, $\mathbf{h}_1 = f_1(\mathbf{x}, \mathbf{w}_1)$. Después $\mathbf{h}_2 = f_2(\mathbf{h}_1, \mathbf{w}_2)$, y así sucesivamente, hasta $\mathbf{h}_l = f_l(\mathbf{h}_{l-1}, \mathbf{w}_l)$. Por último, $y = \Sigma \mathbf{h}_l$.



La traza de evaluación

Esto se llama traza de evaluación del programa. A grandes rasgos, las primeras $l - 1$ operaciones pueden representar varias capas de un modelo diferenciable, la operación l puede ser una pérdida por entrada, por ejemplo la entropía cruzada, y la operación final suma las

pérdidas del minilote. Por tanto, la salida de nuestro programa es siempre un escalar, ya que lo necesitamos para la optimización numérica. Abreviamos el programa anterior como $F(\mathbf{x})$.

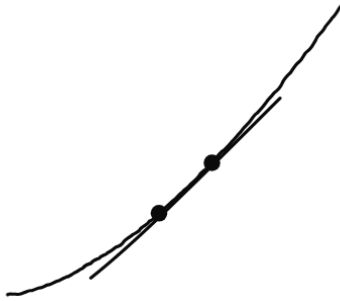
Definición: diferenciación automática. Dado un programa $F(\mathbf{x})$ compuesto por una secuencia de primitivas diferenciables, la diferenciación automática, o AD, es la tarea de calcular de forma simultánea y eficiente todos los jacobianos de los pesos del programa, conociendo el grafo computacional y todos los jacobianos individuales de entrada y de los pesos. En símbolos, $AD(F(\mathbf{x}))$ es el conjunto de los gradientes $\partial_{\mathbf{w}_i}$ y, para $i = 1, \dots, l$.

Como veremos, hay dos grandes clases de algoritmos de AD, llamadas modo directo y modo inverso, que corresponden a un orden distinto en la composición de las operaciones individuales. Veremos también que el modo inverso, llamado retropropagación en la literatura de redes neuronales, es mucho más eficiente en nuestro contexto.

Aunque nos centramos en un escenario simplificado, es relativamente fácil extender nuestra deducción a grafos acíclicos de primitivas, como ya se dijo, y también a situaciones en las que los parámetros se comparten entre capas, lo que se llama compartición de pesos. Veremos un ejemplo de compartición de pesos en el capítulo 13.

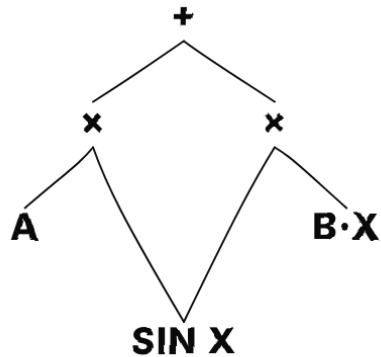
Diferenciación numérica y simbólica.

NUMÉRICA



2 LLAMADAS

SIMBÓLICA



**SE CALCULA 1 VEZ
NP-COMPLETA**

Diferenciación numérica frente a simbólica

Antes de pasar a la AD en modo directo, comentamos la diferencia entre la AD y otras clases de algoritmos para derivar funciones. En primer lugar, podríamos aplicar directamente la definición de gradiente para obtener una aproximación numérica adecuada del gradiente. Este proceso se llama diferenciación numérica. Sin embargo, cada valor escalar que hay que derivar requiere dos llamadas a la función en una implementación ingenua, lo que hace inviable este enfoque salvo para comprobaciones numéricas de la implementación.

En segundo lugar, considere esta función sencilla: $f(x) = a \sin(x) + b x \sin(x)$. Podemos pedir a un motor simbólico que precalcule la ecuación simbólica completa de la derivada. Esto se llama

diferenciación simbólica. En el recuadro 6.1 del libro se le da esta función a la biblioteca de Python SymPy y, al pedirle su derivada respecto de x , devuelve $a \cos(x) + b x \cos(x) + b \sin(x)$.

En una implementación realista, el valor intermedio $h = \sin(x)$ se calcularía una sola vez y se guardaría en una variable intermedia, que además puede reutilizarse en el cálculo correspondiente de la traza del gradiente, y un razonamiento parecido vale para el término $\cos(x)$ de la derivada. Esto es menos trivial de lo que parece: encontrar una implementación óptima del jacobiano que evite todo cálculo innecesario es una tarea NP-completa, conocida como acumulación óptima del jacobiano. Sin embargo, veremos que podemos aprovechar la estructura de nuestro programa para idear una implementación de la AD suficientemente eficiente, mucho mejor que un enfoque simbólico como el anterior, y que de hecho equivale a un enfoque simbólico que admite la presencia de subsecuencias.

Preguntas

1. ¿Por qué el programa de la traza de evaluación termina con una suma, de modo que su salida y es un único número?
 - a. Porque sumar hace más pequeños los jacobianos
 - b. Porque la optimización numérica requiere una salida escalar
 - c. Porque la última capa de un modelo es siempre una suma
 - d. Porque lo necesita la diferenciación en modo directo
2. ¿Qué tiene que calcular exactamente la diferenciación automática, según la definición?

- a. Todos los jacobianos de los pesos del programa, de forma simultánea y eficiente
 - b. El jacobiano de entrada de cada primitiva, uno cada vez
 - c. Una fórmula simbólica de la salida del programa
 - d. El valor del programa para cada entrada
3. ¿Por qué la diferenciación numérica es inviable para un modelo, salvo como comprobación?
- a. Porque solo da la derivada exacta para funciones lineales
 - b. Porque necesita un motor simbólico
 - c. Porque cada valor escalar que hay que derivar necesita dos llamadas a la función en una implementación ingenua
 - d. Porque no puede tratar una salida escalar
4. ¿Cuál es el punto débil de la derivada simbólica de $f(x) = a \sin(x) + b \times \sin(x)$ escrita entera?
- a. Es matemáticamente incorrecta
 - b. Repite trabajo, como $\sin(x)$ y $\cos(x)$, que una implementación realista calcularía una vez y guardaría
 - c. Un ordenador no puede evaluarla
 - d. Solo funciona cuando a y b valen cero

Respuestas

1. **b)** La operación final suma las pérdidas del minilote, así que la salida del programa es siempre un escalar, que necesitamos para la optimización numérica.
2. **a)** La diferenciación automática es la tarea de calcular de forma simultánea y eficiente todos los jacobianos de los pesos del programa,

conociendo el grafo computacional y los jacobianos individuales de entrada y de los pesos.

3. c) Cada valor escalar que hay que derivar requiere dos llamadas a la función en una implementación ingenua, lo que hace inviable el enfoque salvo para comprobaciones numéricas.

4. b) En una implementación realista, el valor intermedio $h = \sin(x)$ se calcularía una sola vez y se guardaría, y lo mismo vale para el término $\cos(x)$; evitar todo cálculo innecesario no es nada trivial.

Ejemplos resueltos

La derivada simbólica en un punto Tome la función del libro $f(x) = a \sin(x) + b x \sin(x)$, cuya derivada simbólica es $a \cos(x) + b x \cos(x) + b \sin(x)$. ¿Cuánto vale la derivada en $x = 0$ cuando $a = 2$ y $b = 3$?

Escriba la derivada que devuelve el motor simbólico: $a \cos(x) + b x \cos(x) + b \sin(x)$.

En $x = 0$, $\cos(0) = 1$ y $\sin(0) = 0$.

El primer término es $a \times 1 = 2$.

El segundo término tiene el factor $x = 0$, así que vale 0; el tercero tiene $\sin(0) = 0$, así que también vale 0.

Sume los tres términos: $2 + 0 + 0 = 2$.

Respuesta: 2

Ejercicios

1. Un modelo tiene 1 000 000 de pesos. Con la diferenciación numérica ingenua, que necesita dos llamadas a la función por cada valor escalar que hay que derivar, ¿cuántas llamadas al programa entero costaría un gradiente?

Pista: Cada peso es un valor escalar que hay que derivar.

Respuesta: 2 000 000 de llamadas: dos por cada uno del millón de pesos, y por eso la diferenciación numérica solo se usa para comprobar una implementación.

2. Escriba la traza de evaluación de un programa con $l = 3$ primitivas, terminando con la suma final. ¿Qué operación podría ser la pérdida por entrada?

Pista: Cada primitiva toma la salida anterior y sus propios pesos.

Respuesta: $h_1 = f_1(x, w_1)$, $h_2 = f_2(h_1, w_2)$, $h_3 = f_3(h_2, w_3)$, $y = \text{suma de } h_3$. Las dos primeras operaciones pueden ser capas del modelo; la tercera, la operación l , puede ser la pérdida por entrada, como la entropía cruzada, y la suma reúne las pérdidas del minilote.

3. Explique con sus propias palabras qué tienen en común la diferenciación automática en modo directo y en modo inverso, y en qué se diferencian, según lo que cuenta esta parte.

Pista: Las dos calculan lo mismo; fíjese en el orden.

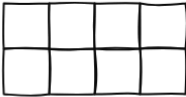
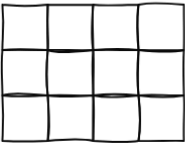
Respuesta: Las dos calculan todos los jacobianos de los pesos del programa a partir de los jacobianos individuales de sus primitivas. Se diferencian en el orden en que se componen las operaciones individuales, y el modo inverso, llamado retropropagación, es mucho más eficiente en este contexto.

Diferenciación en modo directo

Empezamos recordando la regla de la cadena de los jacobianos.

Considere una combinación de dos funciones primitivas, $\mathbf{h} = f_1(\mathbf{x})$ y $y = f_2(\mathbf{h})$. En términos de sus gradientes, tenemos $\partial_{\mathbf{x}}y = \partial_{\mathbf{h}}y \cdot \partial_{\mathbf{x}}\mathbf{h}$.

$$\partial_{\mathbf{x}}y = \partial_{\mathbf{h}}y \cdot \partial_{\mathbf{x}}\mathbf{h}$$

$\partial_{\mathbf{h}}y$		$\partial_{\mathbf{x}}\mathbf{h}$		$\partial_{\mathbf{x}}y$
	$\times$		$=$	
C × B		B × A		C × A

PASO NUEVO

YA CALCULADO

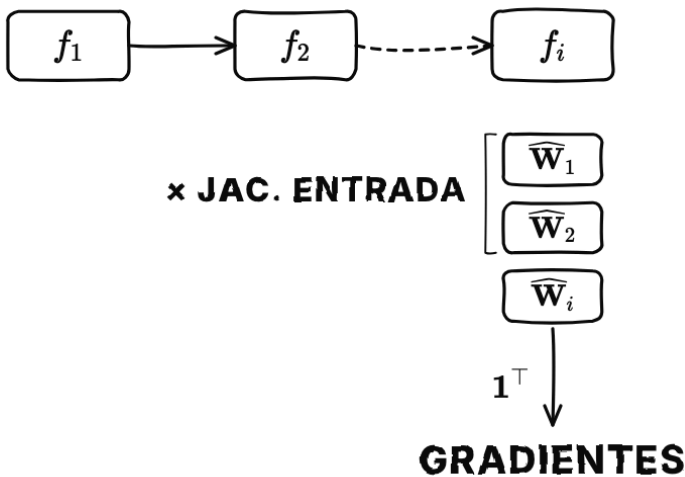
La regla de la cadena como formas de matrices

Si $\mathbf{x}$, $\mathbf{h}$ y y tienen dimensiones a , b y c respectivamente, el jacobiano anterior requiere multiplicar una matriz de $c \times b$, el jacobiano de y respecto de $\mathbf{h}$, por una de $b \times a$, el jacobiano de $\mathbf{h}$ respecto de $\mathbf{x}$. Podemos interpretar la regla así: si ya hemos calculado f_1 y su

jacobiano, una vez que aplicamos f_2 podemos «actualizar» el gradiente multiplicándolo por el jacobiano correspondiente.

Podemos aplicar enseguida esta idea para obtener un algoritmo que funciona, llamado diferenciación automática en modo directo, o F-AD. La idea es que cada vez que aplicamos una función primitiva, inicializamos su jacobiano de los pesos correspondiente, llamado en este contexto la tangente, y al mismo tiempo actualizamos todas las matrices tangentes anteriores. Veamos un ejemplo sencillo resuelto para ilustrar el algoritmo principal.

Considere la primera instrucción de nuestro programa, $\mathbf{h}_1 = f_1(\mathbf{x}, \mathbf{w}_1)$. Como hasta ahora no se ha guardado nada, inicializamos la matriz tangente de $\mathbf{w}_1$ como su jacobiano de los pesos: $\hat{\mathbf{W}}_1 = \partial_{\mathbf{w}_1} \mathbf{h}_1$.



El modo directo lleva consigo las tangentes

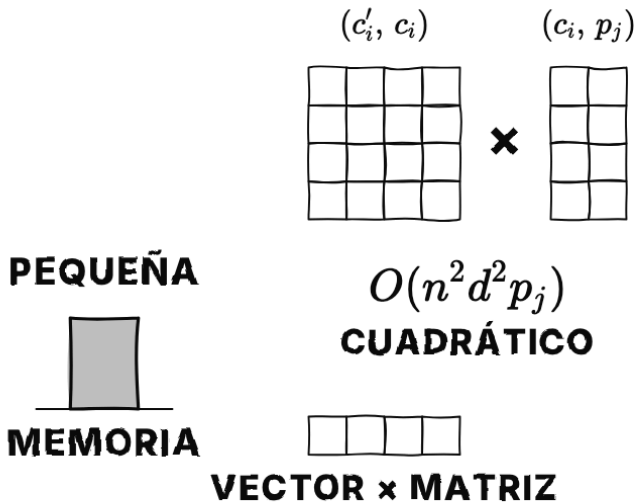
Pasamos ahora a la segunda instrucción, $\mathbf{h}_2 = f_2(\mathbf{h}_1, \mathbf{w}_2)$.

Actualizamos la matriz tangente anterior y al mismo tiempo inicializamos la segunda. La matriz tangente de $\mathbf{w}_1$ se actualiza multiplicándola por el jacobiano de entrada de f_2 : $\hat{\mathbf{W}}_1 \leftarrow [\partial_{\mathbf{h}_1} \mathbf{h}_2] \hat{\mathbf{W}}_1$. La segunda se inicializa como $\hat{\mathbf{W}}_2 = \partial_{\mathbf{w}_2} \mathbf{h}_2$.

La actualización requiere el jacobiano de entrada de la primitiva, mientras que el segundo término requiere el jacobiano de los pesos de la primitiva. En abstracto, considere la i -ésima primitiva genérica, dada por $\mathbf{h}_i = f_i(\mathbf{h}_{i-1}, \mathbf{w}_i)$. Inicializamos la matriz tangente de $\mathbf{w}_i$ y al mismo tiempo actualizamos todas las matrices anteriores. Para cada j menor que i , $\hat{\mathbf{W}}_j \leftarrow [\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i] \hat{\mathbf{W}}_j$. Y la nueva es el jacobiano de los pesos de f_i , $\hat{\mathbf{W}}_i = \partial_{\mathbf{w}_i} \mathbf{h}_i$.

Hay $i - 1$ actualizaciones en la primera fila, una por cada matriz tangente que ya tenemos guardada en memoria, y el jacobiano de entrada de la i -ésima operación es compartido por todas las tangentes anteriores. La última operación del programa es una suma, y el gradiente correspondiente nos da la salida del algoritmo: para cada i , $\nabla_{\mathbf{w}_i} y = \mathbf{1}^T \hat{\mathbf{W}}_i$. Para ser del todo coherentes con la notación, esta salida es un vector fila, mientras que definimos el gradiente como un vector columna; por simplicidad ignoraremos este detalle sutil hasta que llegue el momento de definir los productos vector-jacobiano, más adelante en el capítulo.

¡Listo! Analicemos el algoritmo con más detalle. En primer lugar, todas las operaciones enumeradas pueden intercalarse fácilmente con el programa original, lo que significa que la complejidad en espacio será aproximadamente proporcional a la del programa que estamos derivando.



Ligero en memoria, pesado en aritmética

En el lado negativo, la operación central del algoritmo, la actualización de $\hat{W}_i$, requiere multiplicar dos matrices, de formas genéricas (c'_i, c_i) y (c_i, p_j) , donde c_i y c'_i son las formas de entrada y de salida, y p_j es la forma de w_j . Es una operación extremadamente costosa. Por ejemplo, suponga que entradas y salidas tienen ambas forma (n, d) , donde n es la dimensión del minilote y d representa las características de entrada y de salida. Entonces la multiplicación de matrices tendrá complejidad $O(n^2 d^2 p_j)$, que es cuadrática tanto en el tamaño del minilote como en la dimensión de las características. Esto puede volverse inviable con facilidad, sobre todo para entradas de muchas dimensiones como las imágenes.

Podemos lograr un mejor equilibrio observando que la última operación del algoritmo es un producto matriz-vector más sencillo, consecuencia de tener una salida escalar. Esto se estudia con más detalle en la sección siguiente.

Preguntas

1. En la diferenciación en modo directo, ¿qué les pasa a las matrices tangentes ya guardadas cuando se aplica una primitiva nueva?
 - a. Se tiran y se vuelven a calcular al final
 - b. Se actualizan todas multiplicándolas por el jacobiano de entrada de la primitiva nueva
 - c. Se suman al jacobiano de los pesos de la primitiva nueva
 - d. Se quedan igual hasta la suma final
2. ¿Por qué es modesto el uso de memoria de la diferenciación en modo directo?
 - a. Porque solo guarda el gradiente final
 - b. Porque nunca multiplica matrices
 - c. Porque sus operaciones pueden intercalarse con el programa original, así que su complejidad en espacio es aproximadamente la del programa
 - d. Porque usa un vector de unos en lugar de jacobianos
3. ¿Qué hace costosa la diferenciación en modo directo para entradas grandes como las imágenes?

- a. Su actualización central multiplica dos matrices, con un coste cuadrático tanto en el tamaño del minilote como en el número de características
- b. Tiene que ejecutar el programa dos veces
- c. Necesita un motor simbólico para cada primitiva
- d. Su suma final es un producto de matriz por matriz

Respuestas

1. **b)** Cada vez que se aplica una primitiva, se inicializa su tangente mientras todas las matrices tangentes anteriores se actualizan multiplicándolas por su jacobiano de entrada, que todas comparten.
2. **c)** Todas las operaciones pueden intercalarse fácilmente con el programa original, así que la complejidad en espacio es aproximadamente proporcional a la del programa que se deriva.
3. **a)** La actualización de cada tangente es una multiplicación de matrices con complejidad $O(n^2 d^2 p_j)$, cuadrática en el tamaño del minilote y en la dimensión de las características.

Ejemplos resueltos

Contar las actualizaciones de tangentes Un programa tiene $l = 4$ primitivas. En modo directo, la i -ésima primitiva actualiza las $i - 1$ matrices tangentes ya guardadas. ¿Cuántas actualizaciones de tangentes se hacen en total, y cuántas tangentes se inicializan?

La primera primitiva no encuentra nada guardado: 0 actualizaciones, e inicializa su propia tangente.

La segunda actualiza la única tangente guardada: 1 actualización.

La tercera actualiza dos, la cuarta actualiza tres.

En total $0 + 1 + 2 + 3 = 6$ actualizaciones, cada una una multiplicación de dos matrices.

Cada una de las cuatro primitivas inicializa una tangente, así que se inicializan 4 tangentes.

Respuesta: 6 actualizaciones y 4 tangentes inicializadas

Ejercicios

1. En la regla de la cadena $\partial_x \mathbf{y} = \partial_h \mathbf{y} \cdot \partial_x \mathbf{h}$, suponga que $\mathbf{x}$ tiene dimensión $a = 3$, $\mathbf{h}$ tiene $b = 4$ e $\mathbf{y}$ tiene $c = 2$. ¿Cuáles son las formas de los dos factores y del resultado?

Pista: El libro dice que el producto multiplica una matriz de $c \times b$ por una de $b \times a$.

Respuesta: $\partial_h \mathbf{y}$ es de 2×4 y $\partial_x \mathbf{h}$ es de 4×3 , así que su producto $\partial_x \mathbf{y}$ es de 2×3 ; c por a , como debe ser un jacobiano de $\mathbf{y}$ respecto de $\mathbf{x}$.

2. La actualización en modo directo cuesta $O(n^2 d^2 p_j)$. Si el tamaño del minilote n se duplica mientras d y p_j no cambian, ¿por qué factor crece el coste? ¿Y si se duplican n y d a la vez?

Pista: Fíjese en las potencias de n y d .

Respuesta: Duplicar n multiplica el coste por $2^2 = 4$. Duplicar n y d a la vez lo multiplica por $4 \times 4 = 16$, y por eso el método se vuelve inviable enseguida para entradas grandes.

3. Explique con sus propias palabras por qué el libro espera un mejor equilibrio de la última operación del algoritmo.

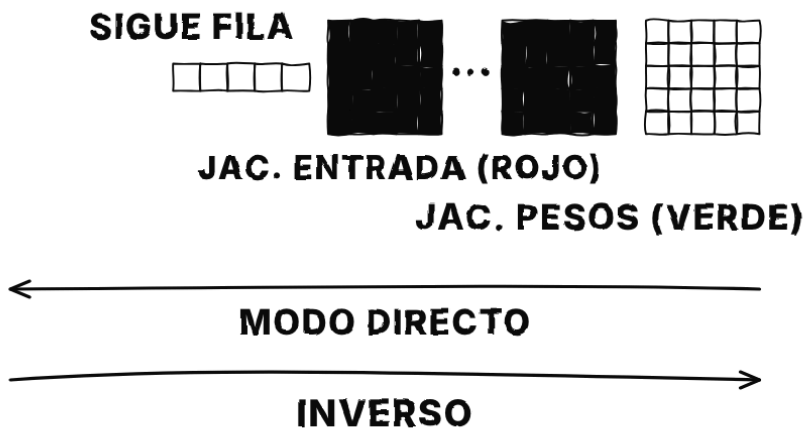
Pista: ¿Qué tiene de especial la salida del programa?

Respuesta: Como el programa termina con una suma y su salida es un escalar, el último paso es un producto matriz-vector más sencillo: el vector de unos traspuesto por una tangente. Un producto por un vector es

mucho más barato que uno entre dos matrices, y la sección siguiente lo aprovecha.

Diferenciación en modo inverso

Para seguir, desplegamos el cálculo de un solo término del gradiente, el que corresponde a la i -ésima matriz de pesos. Se escribe $\nabla_{\mathbf{w}_i} y = \mathbf{1}^T [\partial_{\mathbf{h}_{i-1} \mathbf{h}_i}] \cdots [\partial_{\mathbf{h}_i \mathbf{h}_{i+1}}] [\partial_{\mathbf{w}_i \mathbf{h}_i}]$.



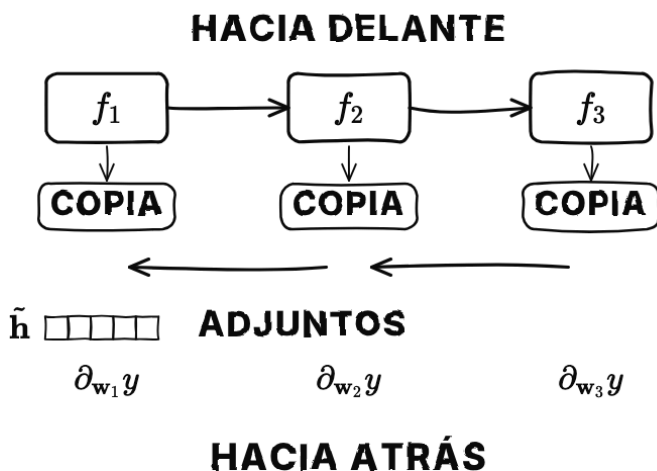
Un gradiente como fila de productos

Recuerde que, notación aparte, esto no es más que una serie, quizá larga, de multiplicaciones de matrices. Incluye un término constante, un vector $\mathbf{1}$ de unos; una serie de jacobianos de entrada, que el libro imprime en rojo; y un jacobiano de los pesos de la matriz de pesos

correspondiente, impreso en verde. Definamos una abreviatura para el término rojo: $\tilde{\mathbf{h}}_i = \mathbf{1}^T \prod_{j=i+1}^l \partial_{\mathbf{h}_{j-1}} \mathbf{h}_j$.

Como la multiplicación de matrices es asociativa, podemos hacer los cálculos de este producto en cualquier orden. En la F-AD avanzábamos de derecha a izquierda, ya que ese es el orden en que se ejecutaban las funciones primitivas. Sin embargo, podemos hacerlo mejor fijándonos en dos aspectos interesantes. Primero, el término más a la izquierda del producto es un producto entre un vector y una matriz, consecuencia de tener una salida escalar, y esto es computacionalmente mejor que un producto entre dos matrices. Su resultado es además otro vector. Segundo, el término abreviado, el producto de todos los jacobianos de entrada desde la capa i hasta la capa l , puede calcularse de forma recursiva, empezando por el último término y multiplicando sucesivamente por los jacobianos de entrada en orden inverso.

Juntando estas observaciones podemos desarrollar un segundo enfoque de la diferenciación automática, que llamamos diferenciación automática en modo inverso, o R-AD, y que se describe a continuación.



Pasada hacia delante, luego pasada hacia atrás

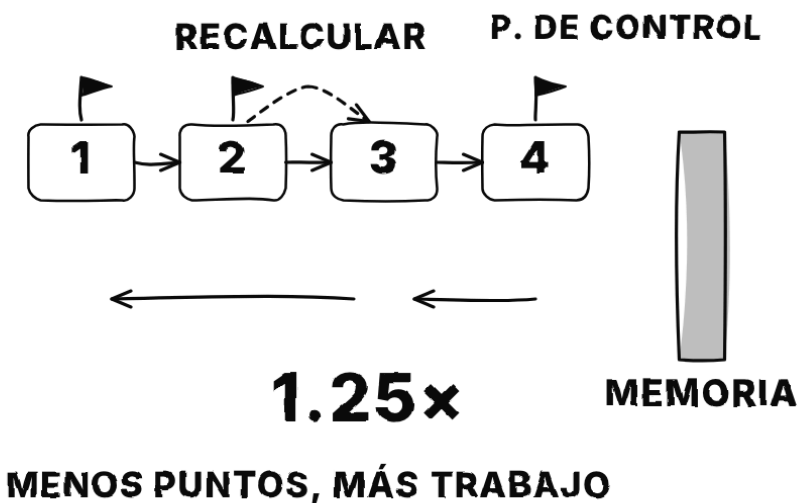
Paso uno: a diferencia de la F-AD, empezamos ejecutando todo el programa que hay que derivar, guardando todas las salidas intermedias. Paso dos: inicializamos un vector $\tilde{\mathbf{h}} = \mathbf{1}^T$, que corresponde al término más a la izquierda del producto. Paso tres: avanzando en orden inverso, es decir, para un índice i que recorre $l, l - 1, l - 2$, y así hasta 1, calculamos primero el gradiente respecto de la i -ésima matriz de pesos como $\partial_{\mathbf{w}_i} y = \tilde{\mathbf{h}} [\partial_{\mathbf{w}_i} \mathbf{h}_i]$, que es el i -ésimo gradiente que necesitamos. Después actualizamos nuestro jacobiano de entrada «retropropagado» como $\tilde{\mathbf{h}} \leftarrow \tilde{\mathbf{h}} [\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i]$.

Los pasos uno a tres describen un programa aproximadamente simétrico al programa original, que llamamos el programa dual, o inverso. Los términos $\tilde{\mathbf{h}}$ se llaman adjuntos, y guardan, uno tras otro,

todos los gradientes de la salida respecto de las variables $\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_i$ de nuestro programa. Compárelo con la F-AD, donde las tangentes representaban en cambio los gradientes de las variables $\mathbf{h}_i$ respecto de los pesos.

En la terminología de las redes neuronales, a veces decimos que el programa original, o primal, es una pasada hacia delante, que no hay que confundir con el modo directo, mientras que el programa inverso es una pasada hacia atrás. A diferencia de la F-AD, en la R-AD el programa primal completo debe ejecutarse antes de poder ejecutar el programa inverso, y necesitamos mecanismos especializados para guardar todas las salidas intermedias y así «desenrollar» el grafo computacional. Cada entorno lo implementa de manera distinta, como se describe a continuación.

Computacionalmente, la R-AD es mucho más eficiente que la F-AD. En particular, las dos operaciones del paso tres de la R-AD son productos vector-matriz, que crecen solo linealmente con todas las magnitudes de forma. La contrapartida es que ejecutar la R-AD requiere mucha memoria, ya que todos los valores intermedios del programa primal deben guardarse con una estrategia adecuada. Pueden usarse técnicas específicas, como los puntos de control del gradiente, para mejorar este equilibrio aumentando los cálculos y reduciendo en parte la memoria necesaria. Esto se hace guardando solo unas pocas salidas intermedias, llamadas puntos de control, y recalculando los valores restantes durante la pasada hacia atrás.



Puntos de control del gradiente

La figura 6.1 muestra un ejemplo de puntos de control del gradiente en cuatro imágenes. En la primera, ejecutamos una pasada hacia delante, pero solo guardamos las salidas del primer, segundo y cuarto bloque, los puntos de control. En la segunda, la pasada hacia atrás, dibujada con flechas rojas, se detiene en el tercer bloque, cuyas activaciones no están disponibles. En la tercera, ejecutamos una segunda pasada hacia delante, desde el punto de control más cercano, para volver a materializar las activaciones. En la cuarta, completamos la pasada. En comparación con una pasada hacia atrás normal, esto requiere 1,25 veces más cálculo. En general, cuantos menos puntos de control se guardan, mayor es el coste computacional de la pasada hacia atrás.

Preguntas

1. ¿Por qué se puede calcular el largo producto del gradiente de izquierda a derecha en lugar de derecha a izquierda?
 - a. Porque la multiplicación de matrices es conmutativa
 - b. Porque la multiplicación de matrices es asociativa
 - c. Porque todos los jacobianos son cuadrados
 - d. Porque el vector de unos puede ignorarse
2. ¿Qué hace más barato avanzar desde la izquierda, empezando por el vector de unos?
 - a. Cada paso es entonces un producto de un vector por una matriz, cuyo resultado es otra vez un vector
 - b. Ya no hacen falta los jacobianos de los pesos
 - c. Ya no hace falta ejecutar el programa
 - d. Los jacobianos se vuelven diagonales
3. ¿Qué precio se paga por la eficiencia del modo inverso?
 - a. Solo da gradientes aproximados
 - b. Necesita mucha memoria, porque hay que guardar todos los valores intermedios del programa primal
 - c. Solo puede tratar programas de una capa
 - d. Su coste es cuadrático en el tamaño del minilote
4. En los puntos de control del gradiente, ¿qué ocurre si se guardan menos puntos de control?
 - a. Se usa menos memoria y la pasada hacia atrás cuesta más cálculo

- b. Se usa más memoria y la pasada hacia atrás es más rápida
- c. No cambia nada, ya que de todas formas se guarda cada valor
- d. Los gradientes se vuelven menos precisos

Respuestas

1. **b)** Como la multiplicación de matrices es asociativa, los cálculos del producto pueden hacerse en cualquier orden.
2. **a)** El término más a la izquierda es un producto entre un vector y una matriz, consecuencia de la salida escalar; esto es computacionalmente mejor que un producto de dos matrices, y su resultado es otro vector.
3. **b)** El programa primal completo debe ejecutarse primero y hay que guardar todos sus valores intermedios, así que el modo inverso requiere mucha memoria.
4. **a)** Solo se guardan unas pocas salidas y el resto se recalcula durante la pasada hacia atrás; cuantos menos puntos de control se guardan, mayor es el coste computacional de la pasada hacia atrás.

Ejemplos resueltos

El coste de los puntos de control En la figura del libro, los puntos de control hacen que la pasada hacia atrás necesite 1,25 veces el cálculo de una pasada hacia atrás normal. Si una pasada hacia atrás normal en cierto modelo tarda 8 segundos, ¿cuánto tarda con estos puntos de control?

Los puntos de control guardan solo algunas salidas, así que las activaciones que faltan se vuelven a calcular durante la pasada hacia atrás.

En el ejemplo del libro, esto hace que el cálculo sea 1,25 veces el de una pasada hacia atrás normal.

Así que el tiempo pasa a ser $1,25 \times 8$ segundos.

Son 10 segundos: 2 segundos más, pagados a cambio de guardar menos salidas intermedias.

Respuesta: 10 segundos

Ejercicios

1. Un programa tiene $l = 3$ primitivas. Enumere en orden los pasos de la diferenciación en modo inverso: qué gradiente se calcula primero y cómo cambia el adjunto $\tilde{h}$ después de cada uno.

Pista: Empiece ejecutando todo el programa, luego ponga $\tilde{h} = 1^T$ y avance hacia atrás desde $i = 3$.

Respuesta: Ejecute el programa y guarde h_1, h_2, h_3 . Ponga $\tilde{h} = 1^T$. Para $i = 3$: el gradiente de w_3 es $\tilde{h}$ por el jacobiano de h_3 respecto de w_3 , y luego $\tilde{h}$ se multiplica por el jacobiano de h_3 respecto de h_2 . Para $i = 2$: el gradiente de w_2 , y luego $\tilde{h}$ se multiplica por el jacobiano de h_2 respecto de h_1 . Para $i = 1$: el gradiente de w_1 . El gradiente del último peso es el primero que sale.

2. Multiplicar un vector fila de longitud b por una matriz de $b \times a$ cuesta $b \cdot a$ multiplicaciones; multiplicar una matriz de $c \times b$ por una de $b \times a$ cuesta $c \cdot b \cdot a$. Con $a = b = c = 100$, compare los dos.

Pista: Calcule los dos productos de las dimensiones.

Respuesta: El producto vector-matriz cuesta $100 \times 100 = 10\,000$ multiplicaciones, y el producto matriz-matriz, $100 \times 100 \times 100 = 1\,000\,000$, cien veces más. Por eso compensa empezar por el vector de unos, como hace el modo inverso.

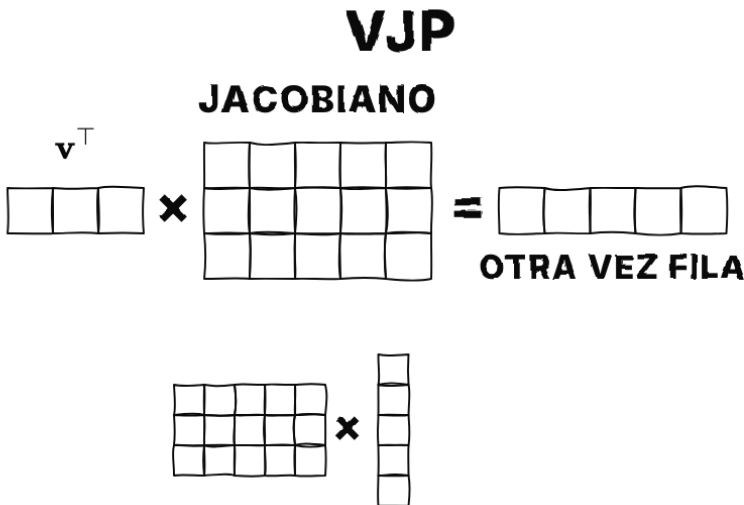
3. Explique con sus propias palabras la diferencia entre lo que guardan las tangentes del modo directo y lo que guardan los adjuntos del modo inverso.

Pista: ¿Gradientes de qué, respecto de qué?

Respuesta: Las tangentes del modo directo son los gradientes de las variables intermedias h respecto de los pesos. Los adjuntos del modo inverso son los gradientes de la salida y respecto de las variables intermedias h_1, h_2 , etcétera, que se guardan uno tras otro mientras se ejecuta el programa inverso.

Productos vector-jacobiano

Consideraciones prácticas.



Una fila por un jacobiano

Si miramos el paso tres del algoritmo R-AD, podemos hacer una observación interesante: la única operación que necesitamos es un producto entre un vector fila $\mathbf{v}$ y un jacobiano de f , ya sea el de entrada o el de los pesos. Llamamos a estas dos operaciones los productos vector-jacobiano, o VJP, de f . En cambio, la F-AD puede formularse por completo en términos de la traspuesta del VJP, llamada producto jacobiano-vector, o JVP; para una salida

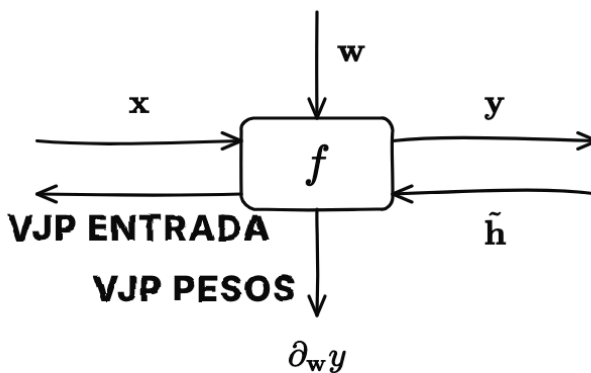
unidimensional, el JVP es la derivada direccional. Por analogía, el VJP representa la aplicación de un mapa lineal ligado a variaciones infinitesimales de la salida de la función. En la definición siguiente restablecemos la coherencia dimensional añadiendo una traspuesta al vector.

Definición: el producto vector-jacobiano. Dada una función $y = f(\mathbf{x})$, con $\mathbf{x} \sim (c)$ e $y \sim (c')$, su VJP es otra función, definida como $\text{vjp}_f(\mathbf{v}) = \mathbf{v}^T \partial f(\mathbf{x})$, donde $\mathbf{v} \sim (c')$. Si f tiene varios parámetros, $f(\mathbf{x}_1, \dots, \mathbf{x}_n)$, podemos definir n VJP individuales, uno para cada uno de $\mathbf{x}_1, \dots, \mathbf{x}_n$.

En particular, en nuestro caso podemos definir dos tipos de VJP, que corresponden a las derivadas parciales de la primitiva respecto de los argumentos de entrada y de pesos. El VJP de entrada es $\text{vjp}_f, \mathbf{x}(\mathbf{v}) = \mathbf{v}^T \partial_{\mathbf{x}} f(\mathbf{x}, \mathbf{w})$. El VJP de los pesos es $\text{vjp}_f, \mathbf{w}(\mathbf{v}) = \mathbf{v}^T \partial_{\mathbf{w}} f(\mathbf{x}, \mathbf{w})$.

Ahora podemos reescribir las dos operaciones del paso tres del algoritmo R-AD como dos llamadas VJP de la función primitiva con los valores adjuntos, ignorando el índice i por legibilidad: el adjunto por el VJP de los pesos, y el adjunto por el VJP de entrada. La primera da el gradiente, $\partial_{\mathbf{w}} y = \text{vjp}_f, \mathbf{w}(\tilde{h})$. La segunda actualiza el adjunto, $\tilde{h} \leftarrow \text{vjp}_f, \mathbf{x}(\tilde{h})$.

Por tanto, podemos implementar todo un sistema de diferenciación automática eligiendo primero un conjunto de operaciones primitivas y ampliándolas después con los VJP correspondientes, sin tener que materializar los jacobianos en memoria en ningún momento. Esto se muestra de forma esquemática en la figura 6.2: para hacer R-AD, las primitivas deben ampliarse con dos operaciones VJP, el VJP de entrada y el VJP de los pesos, y una llamada a cada una basta para hacer la pasada hacia atrás a través de la primitiva.

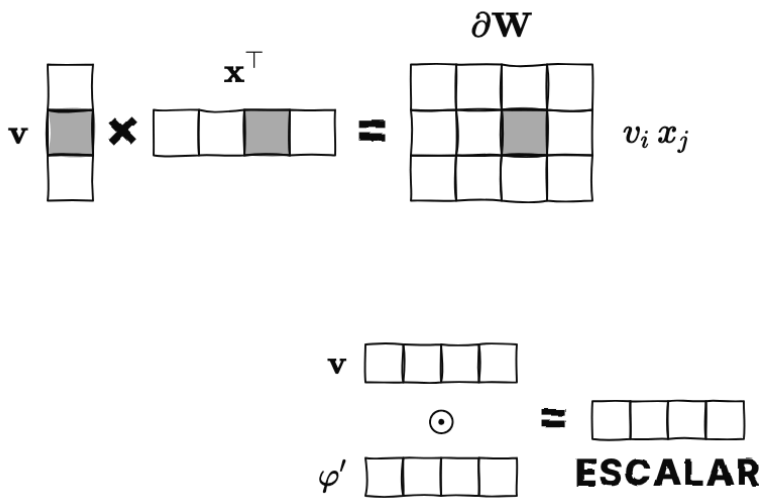


Una primitiva con dos VJP

De hecho, podemos recuperar los jacobianos llamando repetidamente a los VJP con los vectores de la base $e_1, \dots, e_n$, generándolos fila a fila. Para el jacobiano de entrada, por ejemplo, las filas de $\partial_x f(x, w)$ son $\text{vjp_f, x}(e_1)$, $\text{vjp_f, x}(e_2)$, y así hasta $\text{vjp_f, x}(e_n)$.

Para entender por qué esta reformulación puede ser cómoda, veamos los VJP de una capa totalmente conectada, que se compone de proyecciones lineales y no linealidades elemento a elemento. Primero, considere una proyección lineal sencilla sin sesgo, $f(x, W) = Wx$. Aquí el jacobiano de entrada es simplemente W , pero el jacobiano de los pesos es un tensor de rango 3. En comparación, el VJP de entrada no tiene ninguna estructura especial: $\text{vjp_f, x}(v) = v^T W^T = [Wv]^T$. El VJP

de los pesos, en cambio, resulta ser un simple producto exterior, que evita por completo los tensores de rango 3: $\text{vjp}_f, \mathbf{w}(\mathbf{v}) = \mathbf{v}\mathbf{x}^T$.



Producto exterior y escalado

Para obtener este VJP, podemos escribir $y = \mathbf{v}^T \mathbf{W} \mathbf{x} = \sum_i \sum_j W_{ij} v_i x_j$. De aquí obtenemos de inmediato $\partial y / \partial W_{ij} = v_i x_j$, que es la definición elemento a elemento del producto exterior.

Por tanto, cada vez que aplicamos una proyección lineal en la pasada hacia delante, modificamos los gradientes retropropagados por la traspuesta de sus pesos, y hacemos un producto exterior para calcular el gradiente de $\mathbf{W}$.

Considere ahora una función de activación elemento a elemento sin parámetros entrenables, por ejemplo la ReLU: $f(\mathbf{x}) = \phi(\mathbf{x})$, con un

conjunto de pesos vacío. Como no tenemos parámetros entrenables, solo hace falta considerar el VJP de entrada. El gradiente es una matriz diagonal cuyos elementos son las derivadas de ϕ : su i -ésima entrada diagonal es $\phi'(x_i)$. El VJP de entrada es una multiplicación de una matriz diagonal por un vector, lo que equivale a un producto de Hadamard, es decir, a un escalado: $\text{vjp_x}(f, \mathbf{v}) = \mathbf{v} \odot \phi'(\mathbf{x})$. Curiosamente, también en este caso podemos calcular el VJP sin tener que materializar la matriz diagonal completa.

Preguntas

1. ¿Por qué los productos vector-jacobiano son todo lo que la diferenciación en modo inverso necesita de una primitiva?
 - a. Porque el paso tres del algoritmo solo multiplica un vector fila, el adjunto, por un jacobiano de entrada o de los pesos
 - b. Porque los jacobianos de las primitivas son siempre vectores
 - c. Porque la pasada hacia delante calcula de todos modos los jacobianos
 - d. Porque un VJP ocupa menos que la entrada
2. ¿Qué se gana dando a cada primitiva sus dos VJP en lugar de sus jacobianos?
 - a. Los gradientes pasan de aproximados a exactos
 - b. Ya no hay que ejecutar la pasada hacia delante
 - c. Los jacobianos no tienen que materializarse nunca en memoria
 - d. Las primitivas ya no necesitan pesos

3. En una proyección lineal $f(x, W) = Wx$, ¿por qué es cómodo el VJP de los pesos?
- a. Vale cero
 - b. Es un simple producto exterior, $v x^T$, que evita por completo el jacobiano de los pesos de rango 3
 - c. Es igual al jacobiano de entrada
 - d. Necesita la matriz diagonal de derivadas
4. ¿A qué equivale el VJP de entrada de una activación elemento a elemento como la ReLU?
- a. A una multiplicación por la traspuesta de los pesos
 - b. A un producto exterior
 - c. A un escalado de v , elemento a elemento, por las derivadas $\phi'(x)$
 - d. A una suma de todos los elementos de v

Respuestas

1. **a)** La única operación que necesita el paso tres es un producto entre un vector fila y un jacobiano de f , el de entrada o el de los pesos; esos son los dos VJP de f .
2. **c)** Todo un sistema de diferenciación automática puede construirse con primitivas ampliadas con sus VJP, sin tener que materializar los jacobianos en memoria en ningún momento.
3. **b)** El jacobiano de los pesos es un tensor de rango 3, pero el VJP de los pesos resulta ser el producto exterior $v x^T$, que evita por completo los tensores de rango 3.
4. **c)** Su gradiente es una matriz diagonal de las derivadas de ϕ , y multiplicar un vector por ella es un producto de Hadamard, $v \odot \phi'(x)$, un escalado que no necesita ninguna matriz diagonal en memoria.

Ejemplos resueltos

El VJP de los pesos como producto exterior Una proyección lineal $f(x, W) = Wx$ lleva $x = (3, 4, 5)$ a dos salidas, así que W es de 2×3 . El adjunto que llega desde arriba es $v = (1, 2)$. Calcule el VJP de los pesos, es decir, el gradiente de W .

El libro muestra que $\partial y / \partial W_{i,j} = v_i x_j$, así que el VJP de los pesos es el producto exterior $v x^T$.

La fila 1 usa $v_1 = 1$: $(1 \times 3, 1 \times 4, 1 \times 5) = (3, 4, 5)$.

La fila 2 usa $v_2 = 2$: $(2 \times 3, 2 \times 4, 2 \times 5) = (6, 8, 10)$.

El resultado tiene la misma forma de 2×3 que W , y nunca se construyó ningún tensor de rango 3.

Respuesta: La matriz de 2×3 con filas $(3, 4, 5)$ y $(6, 8, 10)$

Ejercicios

1. Una ReLU recibe $x = (1, -2, 0, 5)$; su derivada es 1 donde la entrada es positiva y 0 donde es negativa. El adjunto es $v = (2, -1, 3)$. Calcule el VJP de entrada $v \odot \phi'(x)$.

Pista: Escriba primero $\phi'(x)$ elemento a elemento y luego multiplique elemento a elemento.

Respuesta: $\phi'(x) = (1, 0, 1)$, así que $v \odot \phi'(x) = (2, 0, 3)$. La componente del medio se apaga porque su entrada era negativa.

2. Una primitiva tiene 4 salidas. ¿Cuántas llamadas a su VJP de entrada necesita para reconstruir su jacobiano de entrada completo, y con qué vectores?

Pista: Cada llamada con un vector de la base da una fila.

Respuesta: Cuatro llamadas, con los vectores de la base e_1, e_2, e_3 y e_4 ; cada llamada devuelve una fila del jacobiano de entrada.

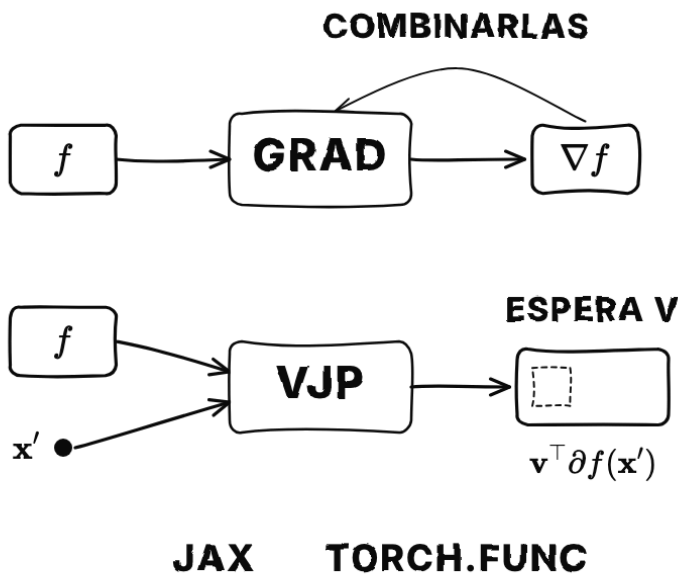
3. Explique con sus propias palabras qué le pasa al gradiente retropropagado, y qué se calcula para los pesos, cada vez que la pasada hacia delante contiene una proyección lineal.

Pista: Hay dos VJP: uno para la entrada y otro para los pesos.

Respuesta: El adjunto que vuelve hacia atrás se multiplica por la traspuesta de los pesos de la capa, mediante el VJP de entrada. Al mismo tiempo, el gradiente de W es el producto exterior del adjunto y la entrada de la capa, mediante el VJP de los pesos.

Implementar un sistema en modo inverso

Hay muchas formas de implementar el sistema R-AD, desde las listas de Wengert, como hace TensorFlow, hasta las transformaciones de código fuente a código fuente. Aquí comentamos brevemente algunas implementaciones habituales en los entornos existentes.



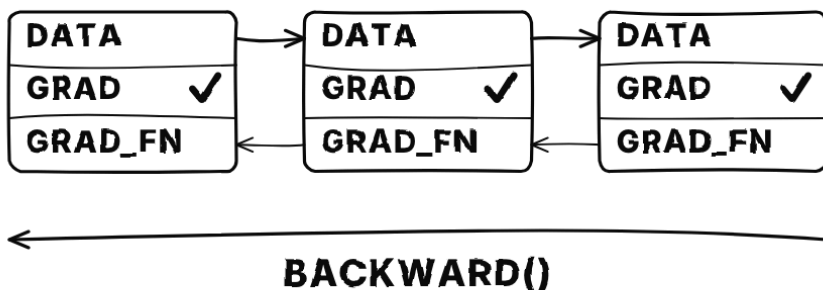
Funciones que fabrican funciones

En primer lugar, describir las primitivas como funciones con dos argumentos, $f(\mathbf{x}, \mathbf{w})$, encaja con entornos funcionales como JAX, donde todo es una función. Considere una función $f(\mathbf{x})$ con una

entrada de dimensión c y una salida de dimensión c' . Desde este punto de vista, un VJP puede implementarse como una función de orden superior con la signatura $(\mathbb{R}^c \rightarrow \mathbb{R}^{c'}) \rightarrow \mathbb{R}^c \rightarrow (\mathbb{R}^{c'} \rightarrow \mathbb{R}^c)$. Es decir, dada una función f y una entrada $\mathbf{x}'$, un VJP devuelve otra función, que puede aplicarse a un vector $\mathbf{v}$ de dimensión c' para devolver $\mathbf{v}^T \partial f(\mathbf{x}')$.

De forma parecida, el gradiente de una función unidimensional puede implementarse como otra función de orden superior, con la signatura $(\mathbb{R}^c \rightarrow \mathbb{R}) \rightarrow (\mathbb{R}^c \rightarrow \mathbb{R}^c)$. Recibe como entrada la función $f(\mathbf{x})$ y devuelve otra función, que calcula $\nabla f(\mathbf{x})$. En JAX, estas ideas están implementadas en funciones como `jax.grad` y `jax.vjp`, y se reproducen en PyTorch en el módulo `torch.func`. En el recuadro 6.2 del libro, por ejemplo, se define una función de suma de cuadrados, `func.grad` la convierte en su función gradiente y, aplicada a un vector aleatorio de diez números, devuelve un gradiente de tamaño diez. La interfaz `torch.func` reproduce la de JAX y, en la práctica, la función puede trazarse, por ejemplo con `torch.compile`, para generar un grafo computacional optimizado. Muchas operaciones, como calcular una hessiana, pueden lograrse combinando con astucia los JVP y los VJP según sus signaturas.

Como ya dijimos, en la práctica nuestros modelos se implementan como composiciones de objetos cuyos parámetros están encapsulados como propiedades. Una posibilidad es «purificar» el objeto, convirtiéndolo en una función pura. En PyTorch, por ejemplo, se extraen los parámetros del modelo en un diccionario y luego se llama a la función hacia delante del modelo mediante `torch.func.functional_call`, pasándole el modelo, los parámetros y la entrada.



MICROGRAD

Tensores que recuerdan su origen

De forma más general, los entornos como PyTorch se amplían con técnicas para tratar este caso directamente, sin introducir operaciones intermedias. En PyTorch, por ejemplo, los objetos tensor se amplían con información sobre la operación que los generó. Siempre que se pide una llamada `backward()` sobre un valor escalar, estas propiedades se usan para recorrer el grafo computacional en sentido inverso, guardando los gradientes correspondientes dentro de los tensores que los requieren.

La figura 6.3 lo muestra en dos mitades. A la izquierda, un tensor de PyTorch lleva información sobre su gradiente, vacío al inicializarse, y sobre la operación que lo creó. A la derecha, durante una pasada hacia atrás, la propiedad `grad_fn` se usa para recorrer el grafo

computacional en sentido inverso, y los gradientes se guardan dentro de la propiedad `grad` del tensor siempre que `requires_grad` se haya puesto explícitamente a verdadero, para no consumir memoria innecesaria.

Esto no es más que una visión general de cómo se implementan estos sistemas en la práctica, y dejamos fuera muchos detalles, para los que remitimos a la documentación oficial. El autor recomienda con entusiasmo intentar implementar desde cero un sistema en modo inverso; en internet hay muchas implementaciones didácticas, como `micrograd`, de Andrej Karpathy.

Preguntas

1. En un marco funcional como JAX, ¿qué recibe y qué devuelve la operación de gradiente?
 - a. Recibe un número y devuelve un número
 - b. Recibe una función f y devuelve otra función, que calcula el gradiente de f en cualquier entrada
 - c. Recibe un tensor y guarda el gradiente dentro de él
 - d. Recibe una lista de operaciones y devuelve sus jacobianos
2. ¿Qué devuelve un VJP cuando se le da una función f y una entrada x' ?
 - a. El jacobiano completo de f en x'
 - b. El valor $f(x')$
 - c. Otra función que, aplicada a un vector v , devuelve v^T por el jacobiano de f en x'

- d. Una lista de los parámetros de f
3. ¿Cómo sabe PyTorch recorrer hacia atrás el grafo computacional cuando se llama a `backward()`?
- a. Cada tensor lleva información sobre la operación que lo creó, y esa información se sigue en sentido inverso
 - b. Deriva simbólicamente el código fuente
 - c. Vuelve a ejecutar la pasada hacia delante con pequeños cambios en cada entrada
 - d. Pide al usuario que enumere las operaciones

Respuestas

1. **b)** El gradiente es una función de orden superior: recibe la función $f(x)$ y devuelve otra función que calcula $\nabla f(x)$.
2. **c)** Dados f y x' , un VJP devuelve otra función que puede aplicarse a un vector v de dimensión c' para obtener $v^T \partial f(x')$.
3. **a)** Los objetos tensor se amplían con información sobre la operación que los generó, y al llamar a `backward()` esas propiedades se usan para recorrer el grafo en sentido inverso.

Ejemplos resueltos

El gradiente de la suma de cuadrados En el recuadro 6.2 del libro la función es $f(x)$ = la suma de los cuadrados de los elementos de x . Su gradiente es $2x$. ¿Qué devuelve la función gradiente para $x = (1, -2, 3)$, y qué forma tiene comparada con x ?

Cada elemento x_k aparece en f solo como x_k^2 , cuya derivada es $2x_k$.

Así que el gradiente es $(2 \times 1, 2 \times (-2), 2 \times 3) = (2, -4, 6)$.

El gradiente tiene un valor por cada elemento de la entrada, así que su forma es la de x , igual que en el recuadro un vector de tamaño 10 da un gradiente de tamaño 10.

Respuesta: $(2, -4, 6)$, con la misma forma que x

Ejercicios

1. Lea con sus propias palabras la firma del gradiente, $(\mathbb{R}^c \rightarrow \mathbb{R}) \rightarrow (\mathbb{R}^c \rightarrow \mathbb{R}^c)$. ¿Por qué su entrada debe ser una función con salida de una sola dimensión?

Pista: Fíjese en lo que entra por la izquierda y en lo que sale por la derecha.

Respuesta: Recibe una función que va de c números a un solo número y devuelve una función que va de c números a c números. El gradiente está definido para una salida escalar: para cada una de las c entradas da una derivada parcial, así que el resultado tiene c valores.

2. Explique por qué un modelo escrito como un objeto con parámetros debe «purificarse» antes de usarlo con herramientas funcionales como `func.grad`.

Pista: Las herramientas funcionales derivan con respecto a los argumentos de una función.

Respuesta: Las herramientas funcionales tratan todo como una función y derivan con respecto a sus argumentos, como en $f(x, w)$. En un objeto, los parámetros están ocultos dentro como propiedades, así que primero se extraen y luego se pasan explícitamente mediante una llamada funcional, lo que convierte el modelo en una función pura de sus parámetros y de su entrada.

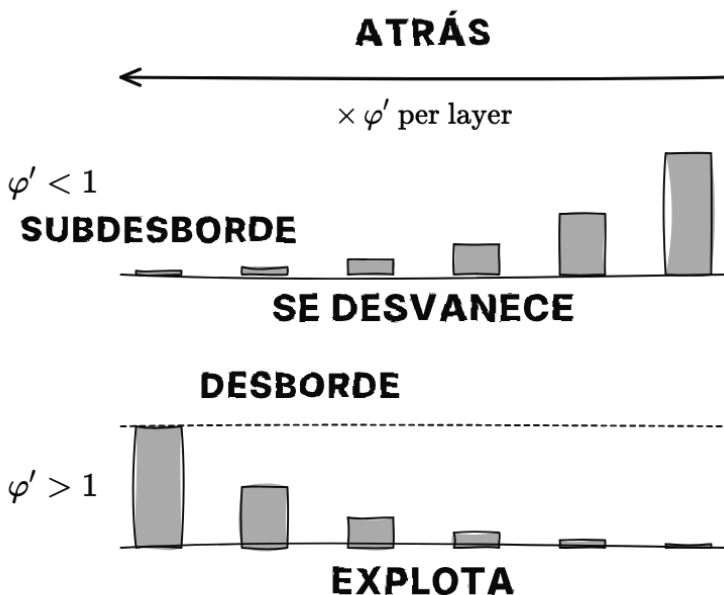
3. En PyTorch, ¿por qué los gradientes se guardan solo en los tensores cuyo `requires_grad` está activado?

Pista: Piense en la memoria.

Respuesta: Guardar un gradiente dentro de cada tensor consumiría memoria sin necesidad; PyTorch guarda gradientes solo donde se piden explícitamente, normalmente en los parámetros que se entrenan.

Elegir una función de activación

Casualmente, ahora podemos justificar por qué la ReLU es una buena elección como función de activación. Una mirada atenta al VJP de entrada de una función de activación nos dice que cada vez que añadimos una función de activación a nuestro modelo, los adjuntos de la pasada hacia atrás se escalan por un factor $\phi'(\mathbf{x})$. En modelos con muchas capas, esto puede dar lugar a dos comportamientos patológicos. Si $\phi' < 1$ en todas partes, existe el riesgo de que el gradiente se reduzca a cero exponencialmente rápido con el número de capas. Esto se llama el problema del desvanecimiento del gradiente. A la inversa, si $\phi' > 1$ en todas partes, aparece el problema contrario, con gradientes que crecen hacia el infinito exponencialmente rápido con el número de capas. Esto se llama el problema de la explosión del gradiente.

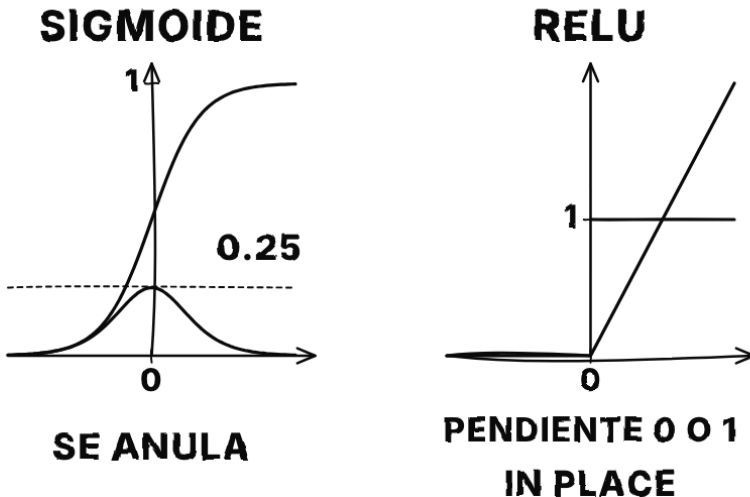


Desvanecimiento y explosión del gradiente

Son problemas serios en la práctica, porque las bibliotecas representan los números de coma flotante con una precisión limitada, normalmente de 32 bits o menos, lo que significa que pueden aparecer enseguida desbordamientos por defecto o por exceso a medida que aumenta el número de capas.

Sorprendentemente, ¡una pila de capas lineales implementada en coma flotante no es del todo lineal, por pequeñas discontinuidades al nivel de la precisión de la máquina! Esto en general no es un problema, pero puede aprovecharse para entrenar redes neuronales profundas totalmente lineales.

Como ejemplo de cómo puede aparecer el desvanecimiento del gradiente, considere la función sigmoide, $\sigma(s)$. Ya mencionamos que era una función de activación habitual en el pasado, porque es una aproximación suave de la función escalón. También sabemos que $\sigma'(s) = \sigma(s)(1 - \sigma(s))$. Junto con el hecho de que $\sigma(s)$ está entre 0 y 1, obtenemos que $\sigma'(s)$ está entre 0 y 0,25. Por tanto, la sigmoide es una candidata de primera a los problemas de desvanecimiento del gradiente. La figura 6.4 muestra, a la izquierda, la sigmoide y su derivada, y a la derecha, la ReLU y su derivada.



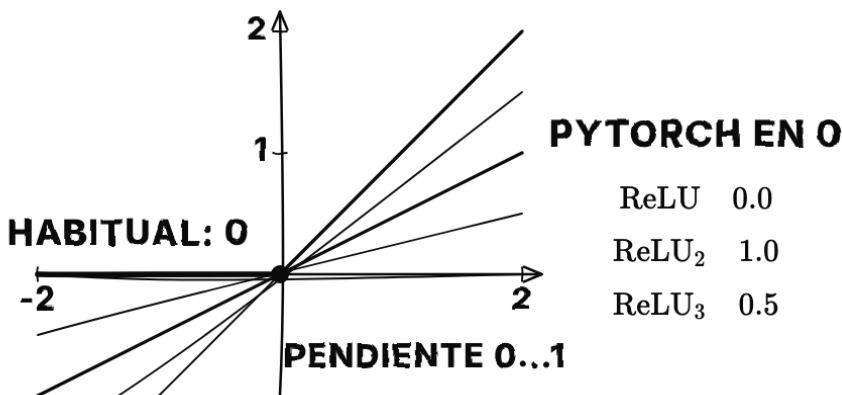
Pendientes de la sigmoide y de la ReLU

Diseñar una función de activación que nunca presente desvanecimiento ni explosión del gradiente no es trivial, ya que la única función con $\varphi'(s) = 1$ en todas partes es la función identidad.

Necesitamos entonces una función que sea «lo bastante lineal» para evitar los problemas del gradiente, pero «lo bastante no lineal» para separar las capas lineales. La ReLU resulta ser una buena candidata, ya que su derivada es 0 para s menor que 0, y 1 para s mayor que 0. El gradiente o bien se anula, lo que induce dispersión en el cálculo, o bien se multiplica por 1, lo que evita los problemas de escala; esto se muestra a la derecha de la figura 6.4.

Como nota al margen, el gradiente de la ReLU es idéntico tanto si lo calculamos a partir de la entrada de la capa ReLU como de su salida, ya que solo enmascaramos los valores negativos y dejamos intactos los positivos. Por tanto, otra ventaja de usar la ReLU como función de activación es que podemos ahorrar algo de memoria al hacer R-AD, sobrescribiendo la entrada de la capa en la pasada hacia delante sin afectar a la corrección del procedimiento de AD. Esto se hace en PyTorch, por ejemplo, activando el parámetro in-place.

Subdiferenciabilidad y diferenciación automática.



Muchas pendientes en la esquina

Hay un pequeño detalle que hemos evitado discutir hasta ahora: la ReLU no es diferenciable en 0, lo que hace que la red en su conjunto no sea suave. ¿Qué ocurre en este caso? La respuesta «pragmática» es que, al minimizar con descenso de gradiente estocástico desde una inicialización aleatoria distinta de cero, la probabilidad de acabar exactamente en $s = 0$ es prácticamente nula, mientras que el gradiente está definido en $\text{ReLU}(\epsilon)$ para cualquier ϵ distinto de cero.

Para una respuesta más técnica, podemos introducir el concepto de subgradiente de una función. Definición: dada una función convexa $f(x)$, un subgradiente en x es un número z tal que, para todo y , $f(y) \geq f(x) + z(y - x)$.

Observe la semejanza con la definición de convexidad: un subgradiente es la pendiente de una recta «tangente» a $f(x)$ tal que toda la función f queda por encima de ella. Si f es diferenciable en x , solo existe una recta así, y su pendiente es la derivada de f en x . En un punto no suave existen varios subgradientes, que forman un conjunto llamado subdiferencial de f en x : el conjunto de todos los z que son subgradientes de $f(x)$.

Con esta definición en la mano, podemos completar nuestro análisis del gradiente de la ReLU sustituyendo el gradiente por su subdiferencial en 0 . Para s menor que 0 , el subdiferencial contiene solo el 0 ; para s mayor que 0 , contiene solo el 1 ; y en $s = 0$ es todo el intervalo de 0 a 1 . Por tanto, cualquier valor entre 0 y 1 es un subgradiente válido en 0 , y la mayoría de las implementaciones eligen en la práctica $\text{ReLU}'(0) = 0$. Elegir subgradientes en cada paso de un procedimiento iterativo de descenso se llama descenso por subgradiente.

De hecho, la situación es aún más delicada, porque el subgradiente no tiene por qué estar definido para funciones no convexas. En ese caso se puede recurrir a generalizaciones que relajan la definición anterior a un entorno local de x , como el subdiferencial de Clarke. La subdiferenciabilidad también puede crear problemas en la AD, donde distintas implementaciones de la misma función pueden dar subgradientes distintos, y posiblemente no válidos, y hay que considerar versiones más refinadas de la regla de la cadena para una demostración formal. Considere este ejemplo: defina dos funciones, $\text{ReLU}_2(s) = \text{ReLU}(-s) + s$ y $\text{ReLU}_3(s) = 0,5(\text{ReLU}(s) + \text{ReLU}_2(s))$. Ambas equivalen a la ReLU, pero en PyTorch una pasada hacia atrás en 0 devuelve $0,0$ para la ReLU, $1,0$ para ReLU_2 y $0,5$ para ReLU_3 .

De la teoría a la práctica.

Si siguió los ejercicios del capítulo 5, ya vio una aplicación de la R-AD tanto en PyTorch como en JAX, y este capítulo, en especial la sección sobre la implementación de un sistema en modo inverso, debería haber aclarado cómo están implementadas.

Es buena idea intentar reimplementar un sistema sencillo de R-AD, parecido al de PyTorch. Por ejemplo, centrándose en magnitudes escalares, el repositorio `micrograd` es una implementación didáctica muy buena. El único detalle que no tratamos es que, cuando se pasa a un grafo acíclico general, ordenar las variables del grafo computacional antes de la pasada hacia atrás es esencial para no crear caminos de retropropagación erróneos. En `micrograd`, esto se consigue con una ordenación topológica de las variables, poco costosa.

También es interesante intentar implementar en PyTorch una primitiva nueva, en el sentido usado en este capítulo, lo que exige especificar su pasada hacia delante junto con sus VJP. Un ejemplo puede ser una de las funciones de activación entrenables del capítulo 5. Es un ejercicio didáctico, en el sentido de que puede implementarse de forma equivalente creando una subclase de `nn.Module` y dejando que el motor de AD de PyTorch deduzca la pasada hacia atrás.

Todos estos pasos también pueden reproducirse en JAX. Implemente una versión didáctica de JAX con `autodidax`. Escriba una primitiva nueva implementando el VJP correspondiente. Y lea el `JAX Autodiff Cookbook` para descubrir usos avanzados del motor de diferenciación automática, como las derivadas de orden superior, las hessianas y más.

Preguntas

1. ¿Por qué una función de activación cuya derivada es menor que 1 en todas partes puede hacer fracasar el entrenamiento de un modelo profundo?
 - a. Porque los adjuntos se escalan por ϕ' en cada capa, así que el gradiente puede reducirse a cero exponencialmente con el número de capas
 - b. Porque la pasada hacia delante se vuelve lineal
 - c. Porque los pesos crecen hacia el infinito
 - d. Porque el subgradiente deja de estar definido
2. ¿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 está entre 0 y 0,25
 - c. Porque no es diferenciable en 0
 - d. Porque es la función identidad
3. ¿Por qué puede sobrescribirse la entrada de la ReLU con su salida durante la pasada hacia delante sin estropear el modo inverso?
 - a. Porque la ReLU no tiene pasada hacia atrás
 - b. Porque su gradiente es el mismo tanto si se calcula a partir de la entrada como de la salida, ya que solo se enmascaran los valores negativos
 - c. Porque PyTorch vuelve a calcular todas las entradas de todos modos

d. Porque la salida de la ReLU es siempre cero

4. ¿Cuál es el subdiferencial de la ReLU en 0?

a. Solo el valor 0

b. Solo el valor 1

c. Todo el intervalo de 0 a 1

d. Está vacío

Respuestas

1. a) Cada activación escala los adjuntos de la pasada hacia atrás por $\phi'(x)$; si $\phi' < 1$ en todas partes, el gradiente corre el riesgo de reducirse a cero exponencialmente rápido con el número de capas: el problema del desvanecimiento del gradiente.

2. b) Como $\sigma'(s) = \sigma(s)(1 - \sigma(s))$ y $\sigma(s)$ está entre 0 y 1, la derivada está entre 0 y 0,25, así que cada capa sigmoide reduce el gradiente.

3. b) La ReLU solo enmascara los valores negativos y deja intactos los positivos, así que su gradiente es idéntico tanto si se toma de la entrada como de la salida; esto ahorra algo de memoria.

4. c) En $s = 0$ cualquier pendiente entre 0 y 1 da una recta por debajo de la ReLU, así que el subdiferencial es el intervalo de 0 a 1; la mayoría de las implementaciones eligen 0.

Ejemplos resueltos

Diez capas sigmoide En un modelo con 10 capas sigmoide, cada capa escala el adjunto como mucho por 0,25, el valor máximo de σ' . ¿Cuál es, como mucho, el factor por el que se habrá reducido el gradiente tras volver a través de las 10 capas?

El libro muestra que $\sigma'(s)$ está entre 0 y 0,25.

Cada capa multiplica el adjunto que se propaga hacia atrás por σ' , así que como mucho por 0,25.

Diez capas multiplican como mucho por $0,25^{10}$.

$0,25^{10} = 1 / 4^{10} = 1 / 1\,048\,576$, unos 0,00000095.

Incluso en el mejor caso, el gradiente se queda en una millonésima de su tamaño: un gradiente que se desvanece.

Respuesta: Como mucho unos 0,00000095, una parte entre 1 048 576

Ejercicios

1. En cierta entrada la sigmoide da $\sigma(s) = 0,9$. Calcule $\sigma'(s)$ con la fórmula $\sigma'(s) = \sigma(s)(1 - \sigma(s))$.

Pista: Multiplique $\sigma(s)$ por uno menos $\sigma(s)$.

Respuesta: $\sigma'(s) = 0,9 \times 0,1 = 0,09$, muy por debajo del máximo de 0,25.

2. Con la definición $f(y) \geq f(x) + z(y - x)$ para todo y , compruebe si $z = 0,3$ y $z = 1,2$ son subgradientes de la ReLU en $x = 0$.

Pista: En $x = 0$ la condición queda $\text{ReLU}(y) \geq z \cdot y$; pruebe un y positivo y uno negativo.

Respuesta: Para $z = 0,3$: si $y > 0$, se cumple $y \geq 0,3y$; si $y < 0$, se cumple $0 \geq 0,3y$; así que 0,3 es un subgradiente. Para $z = 1,2$: con $y = 1$, $1 \geq 1,2$ es falso, así que 1,2 no lo es. Esto concuerda con que el subdiferencial sea el intervalo de 0 a 1.

3. La $\text{ReLU}_2(s) = \text{ReLU}(-s) + s$ del libro es igual a la ReLU.

Suponiendo que PyTorch toma como 0 la derivada de la ReLU en 0, explique por qué una pasada hacia atrás en 0 devuelve 1,0 para ReLU_2 y 0,5 para $\text{ReLU}_3(s) = 0,5(\text{ReLU}(s) + \text{ReLU}_2(s))$.

Pista: Derive cada término como lo hace el motor de AD, usando $\text{ReLU}'(0) = 0$.

Respuesta: Para ReLU_2 , el motor obtiene $-\text{ReLU}'(0) + 1 = 0 + 1 = 1$.

Para ReLU_3 obtiene $0,5 \times (\text{ReLU}'(0) + \text{ReLU}_2'(0)) = 0,5 \times (0 + 1) = 0,5$.

Las tres funciones son iguales, pero el motor devuelve tres subgradientes distintos.