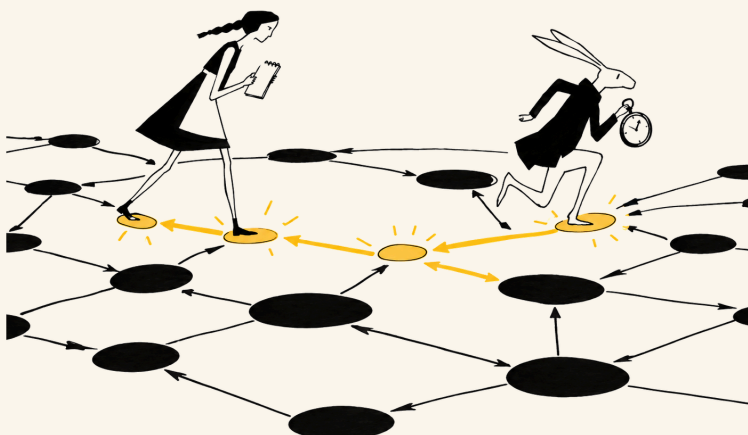


Las aventuras de Alicia en el País de las Maravillas Diferenciables



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

Una receta hecha de pasos pequeños

Por qué los caminos obvios son demasiado lentos

Volver desde la puntuación

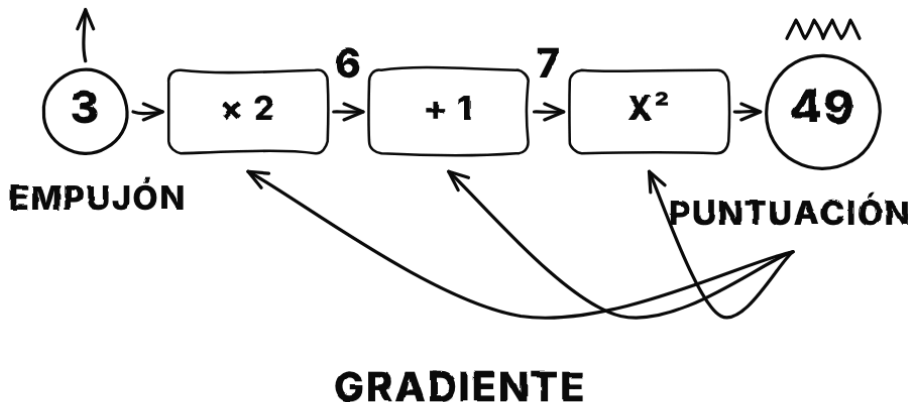
Hacerle una sola pregunta a la tabla

Cómo vuelven atrás los entornos, y por qué ganó la ReLU

La esquina de la ReLU

Una receta hecha de pasos pequeños

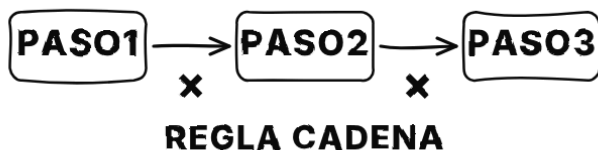
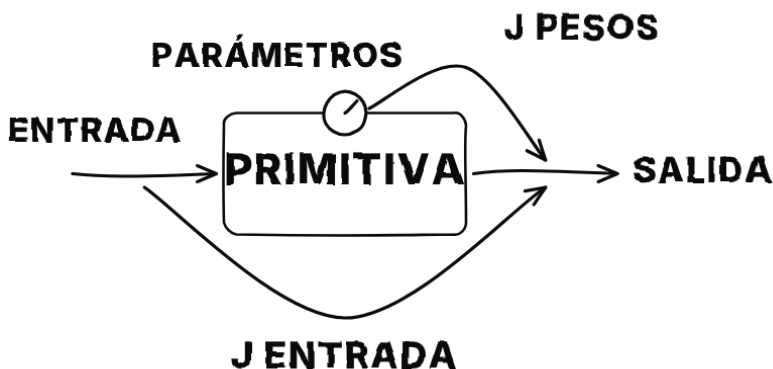
Tome una calculadora y un trozo de papel, y haga una pequeña cuenta por etapas. Empiece con el número tres, duplíquelo, súmele uno y luego eleve el resultado al cuadrado. Obtiene cuarenta y nueve, y cada etapa fue un paso diminuto que se sabe de memoria. Ahora hágase una pregunta más difícil: si sube un pelo el tres inicial, ¿cuánto se mueve el cuarenta y nueve, y qué etapa es responsable de cuánto? Esa pregunta, hecha a una receta mucho más larga, es de lo que trata este capítulo.



Una receta de pasos pequeños

Cuando se entrena un asistente de chat o la aplicación de fotos de un teléfono, el programa que hay detrás es una receta así, solo que con millones de etapas y millones de números ajustables. Al final de la receta hay una única puntuación que dice lo mal que lo hizo el modelo con un lote de ejemplos; el libro la llama la pérdida. Entrenar significa girar un poco cada número ajustable en la dirección que baja esa puntuación, y para eso necesitamos saber, para cada número, con qué fuerza reacciona la puntuación a él. Ese conjunto de reacciones es el gradiente, y calcularlo a máquina, para cualquier programa, se llama diferenciación automática. El método que construye este capítulo es el que usan todos los entornos modernos; quienes trabajan con redes neuronales lo llaman retropropagación, y los informáticos, diferenciación automática en modo inverso.

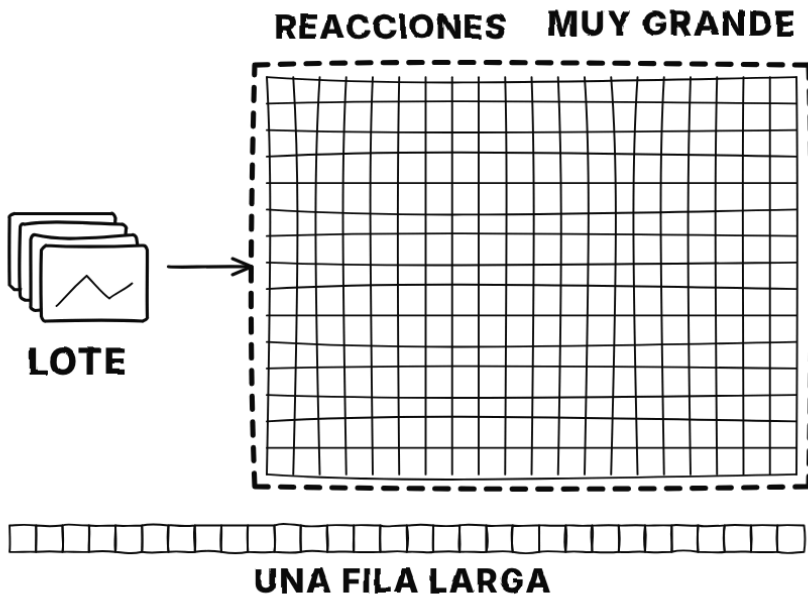
El autor nos pide imaginar un programa como la lista de operaciones pequeñas que realiza de verdad al ejecutarse, una tras otra. El libro llama primitiva a cada operación pequeña, y grafo computacional a toda la cadena que forman. Una primitiva recibe algo, usa algunos números ajustables propios, los parámetros, y entrega algo. Qué cuenta como una primitiva lo decidimos nosotros: una sola multiplicación de tablas de números, una capa entera de una red o incluso un modelo más pequeño completo pueden tratarse como un solo paso. Es la misma costumbre que tienen los programadores en todas partes: construir cosas grandes con bloques que a su vez están hechos de bloques.



Una primitiva, dos reacciones

El libro solo pide una cosa a cada primitiva. Debemos saber cómo reacciona su salida cuando se empuja su entrada, y cómo reacciona su salida cuando se empujan sus propios parámetros. Estas dos tablas de reacciones se llaman el jacobiano de entrada y el jacobiano de los pesos, y son el único conocimiento local que necesita todo el método. A primera vista el problema parece entonces resuelto, porque el cálculo del instituto ya tiene la regla de la cadena: la reacción de una cadena de pasos es el producto de las reacciones de cada paso. La trampa, y el verdadero tema del capítulo, es hacer esa multiplicación lo bastante barata para un modelo del tamaño de la red de una aplicación de fotos.

Algunos pasos tienen una esquina aguda donde la pendiente no está definida, como la popular ReLU, que deja pasar los números positivos y convierte en cero los negativos. El libro promete ocuparse de esas esquinas más adelante en el capítulo; las operaciones que de verdad dan saltos, como sacar una muestra aleatoria, quedan para su segundo volumen.



Una tabla demasiado grande para guardarla

Aquí va una primera pista de por qué importa que sea barato. Una aplicación de fotos nunca procesa una imagen cada vez durante el entrenamiento; toma un lote, digamos unas decenas de imágenes, cada una descrita por miles de números. La tabla de reacciones de una sola capa tiene entonces que conectar cada número de cada imagen que entra con cada número de cada imagen que sale. El libro lo

simplifica colocando todos esos números en una sola fila larga, de modo que «el tamaño de la entrada» siempre significa todo a la vez, lote incluido. Contada así, una tabla de este tipo se vuelve tan enorme que podemos saber perfectamente cómo calcularla y aun así no querer guardarla en memoria en absoluto. Retenga esa idea; es la clave del método ingenioso que viene después.

Por último, el autor señala que los entornos escriben estos pasos en dos estilos. En JAX todo es una función: a un paso se le entregan su entrada y sus parámetros, exactamente como lo escribe el libro. En PyTorch o TensorFlow, una capa es un objeto que guarda dentro sus propios parámetros, como un electrodoméstico con sus ajustes incorporados, y usted simplemente lo enciende con la entrada. El estilo de electrodoméstico es práctico, por ejemplo porque la capa puede esperar a crear sus parámetros hasta ver por primera vez lo grande que es su entrada, pero añade una pequeña capa de traducción entre la forma de escribir del libro y el código real. Esa diferencia de estilo vuelve a aparecer en cómo calculan los gradientes los dos tipos de entorno.

Para recordar

Entrenar un modelo de IA significa empujar millones de números ajustables para bajar una puntuación final, y la diferenciación automática es el método a máquina para averiguar cómo reacciona esa puntuación a cada número. Un programa se ve como una cadena de pasos pequeños llamados primitivas, cada uno con una entrada, sus propios parámetros y una salida. Para cada paso solo necesitamos saber cómo reacciona su salida a su entrada y a sus parámetros: su jacobiano de entrada y su jacobiano de los pesos. La regla de la cadena dice cómo combinar esas reacciones, pero las tablas en juego pueden ser tan enormes que el verdadero reto es combinarlas de forma barata. JAX escribe los pasos como funciones simples, mientras que PyTorch mete cada capa con sus parámetros en un objeto, y eso determina cómo calcula cada uno los gradientes.

Preguntas

1. ¿Por qué necesita el entrenamiento de un modelo saber cómo reacciona la puntuación final a cada número ajustable?
 - a. Porque cada número se gira un poco en la dirección que baja la puntuación
 - b. Porque la puntuación debe mostrarse por separado para cada número
 - c. Porque los números que no afectan a la puntuación se borran antes de entrenar

- d. Porque la regla de la cadena solo funciona cuando se conocen todos los números
2. ¿Qué pide saber el método de cada paso del programa?
- a. Solo la salida del paso para unas pocas entradas de prueba
 - b. Cómo reacciona la salida del paso a los empujones de su entrada y de sus propios parámetros
 - c. Cuánto tarda el paso en ejecutarse en un teléfono
 - d. Cómo reacciona la puntuación final al paso, calculado de antemano
3. Si la regla de la cadena ya dice cómo combinar las reacciones de los pasos, ¿qué queda por resolver?
- a. Encontrar una regla nueva, porque la regla de la cadena falla en programas largos
 - b. Decidir qué pasos cuentan como primitivas, ya que solo una elección es correcta
 - c. Hacer la combinación de forma barata, porque las tablas de reacciones pueden ser enormes
 - d. Quitar los pasos con esquinas agudas, como la ReLU
4. ¿En qué se diferencia una capa al estilo de PyTorch de un paso escrito al estilo de JAX?
- a. La capa de PyTorch no tiene ningún parámetro
 - b. El paso de JAX guarda sus parámetros ocultos dentro de un objeto
 - c. La capa de PyTorch guarda dentro sus parámetros, mientras que el paso de JAX es una función a la que se le dan la entrada y los parámetros

d. No hay diferencia; los dos nombres describen el mismo estilo

Respuestas

1. **a)** El entrenamiento gira un poco cada número ajustable en la dirección que baja la puntuación, y para eso hace falta la reacción de la puntuación a cada número, el gradiente.

2. **b)** De cada primitiva solo necesitamos su jacobiano de entrada y su jacobiano de los pesos, las reacciones de su salida a su entrada y a sus parámetros; todo lo demás se construye a partir de ellos.

3. **c)** La regla de la cadena fija qué hay que calcular; el verdadero reto del capítulo es hacer esa multiplicación lo bastante barata para un modelo del tamaño de la red de una aplicación de fotos.

4. **c)** En JAX todo es una función que recibe su entrada y sus parámetros, mientras que en PyTorch una capa es un objeto con los parámetros incorporados, como un electrodoméstico que se enciende con la entrada.

Ejemplos resueltos

Quién tiene la culpa en una receta corta Empiece con 3, duplíquelo, súmele 1 y eleve el resultado al cuadrado, lo que da 49. Si el 3 inicial sube una cantidad diminuta, ¿aproximadamente cuántas veces esa cantidad se mueve el 49?

Mire cada paso por separado: duplicar hace cualquier empujón 2 veces mayor.

Sumar 1 deja pasar el empujón sin cambios, así que lo multiplica por 1.

Elevar al cuadrado cerca de 7 hace un empujón unas 14 veces mayor, porque 7 al cuadrado crece cerca del doble de 7 por cada paso pequeño.

La regla de la cadena dice que las reacciones de los pasos se multiplican: 2 por 1 por 14.

Respuesta: Unas 28 veces el empujón; empezando en 3,01 se obtiene unos 49,28.

Ejercicios

1. Una aplicación de fotos se entrena con lotes de 32 imágenes, y una capa recibe 1000 números por imagen y entrega 1000 números por imagen. Colocados en una sola fila larga, como hace el libro, ¿cuántas casillas tiene la tabla que conecta cada número que entra con cada número que sale?

Pista: Cuente primero todos los números que entran para todo el lote, luego todos los que salen; la tabla tiene una casilla por cada pareja.

Respuesta: 32 por 1000 son 32 000 números de entrada y 32 000 de salida, así que la tabla tiene 32 000 por 32 000, unos mil millones de casillas. Por eso se puede saber calcular una tabla así y aun así no querer guardarla.

2. Explique con sus propias palabras por qué el libro permite que una capa entera, o incluso un modelo más pequeño completo, cuente como una sola primitiva.

Pista: Piense en cómo se construyen los programas con bloques que a su vez están hechos de bloques.

Respuesta: El método solo necesita saber, para cada paso, cómo reacciona su salida a su entrada y a sus parámetros. Cualquier bloque del que sepamos eso puede tratarse como un solo paso, sea una sola multiplicación, una capa o un modelo entero, igual que los programadores construyen cosas grandes con bloques más pequeños.

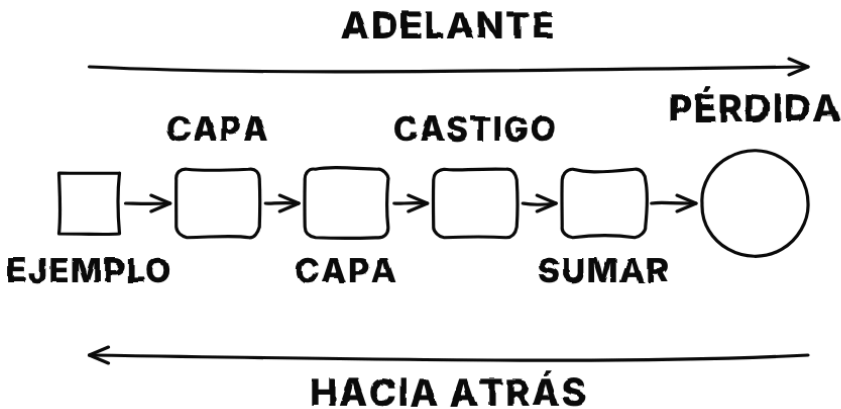
3. Un amigo dice que la ReLU no se puede usar en este método porque tiene una esquina aguda donde su pendiente no está definida. ¿Qué dice esta parte sobre esos pasos?

Pista: Separe los pasos con una sola esquina aguda de las operaciones que de verdad dan saltos.

Respuesta: Los pasos como la ReLU, que son continuos pero tienen una esquina, pueden incorporarse al método; el capítulo promete mostrar cómo más adelante. Las operaciones que de verdad dan saltos, como sacar una muestra aleatoria, necesitan otros trucos y quedan para el segundo volumen del libro.

Por qué los caminos obvios son demasiado lentos

Abra la calculadora de su teléfono y pruebe la forma más directa de averiguar cómo reacciona un resultado a un empujón. Haga la receta de antes, tres duplicado, más uno, al cuadrado, y anote cuarenta y nueve. Luego hágala otra vez desde tres coma cero uno y anote el nuevo resultado, algo más de cuarenta y nueve coma dos. La diferencia, dividida por el tamaño del empujón, le da la reacción, unos veintiocho. Funciona, y le ha costado dos ejecuciones de toda la receta aprender algo sobre un solo número inicial.



La cadena termina en un número

Antes de juzgar ese método, el libro fija con precisión cómo es un programa de entrenamiento. Imagine una fila de puestos en una cadena de montaje. El ejemplo en bruto entra en el primer puesto, cada puesto es una primitiva con sus propios parámetros, y cada uno pasa su resultado al siguiente. Los primeros puestos son las capas del modelo; el último puesto de verdad compara la respuesta del modelo para cada ejemplo con la correcta y pone a cada ejemplo una penalización, como la entropía cruzada que quizá encuentre cuando se entrenan modelos de chat; y un paso final simplemente suma las penalizaciones de todo el lote. Así que la cadena termina siempre en un único número, la pérdida, porque un solo número es lo que un optimizador puede empujar cuesta abajo. El libro llama traza de evaluación del programa a esta secuencia de puestos escrita entera.

Con esa imagen, la tarea recibe un nombre preciso. La diferenciación automática consiste en calcular, todo a la vez y de forma eficiente, cómo reacciona ese número final a los parámetros de cada puesto, usando solo lo que sabemos de cada puesto por separado: su jacobiano de entrada y su jacobiano de los pesos. Hay dos grandes familias de métodos para hacerlo, que solo se diferencian en el orden en que se combinan las reacciones de los puestos. Una recorre la cadena en el sentido en que funciona, el modo directo; la otra vuelve desde el final, el modo inverso, que quienes trabajan con redes neuronales llaman retropropagación. El libro anuncia ya, y lo muestra pronto, que el modo inverso es mucho más barato para programas como los nuestros. Un modelo real no siempre es una línea recta, y algunos modelos reutilizan los mismos parámetros en varios puestos, como hacen los modelos recurrentes de un capítulo posterior; el mismo razonamiento se estira para cubrir ambos casos.

Volvamos ahora a la calculadora. Lo que usted hizo se llama diferenciación numérica: empujar un número, volver a ejecutarlo todo y comparar. Para un número son dos ejecuciones; para un asistente de chat con miles de millones de parámetros son miles de millones de ejecuciones del modelo entero para un solo paso de entrenamiento. Eso no sirve como método de entrenamiento, y el libro lo reserva para una sola tarea modesta: comprobar que un método más ingenioso da las respuestas correctas.

3

3.01

RECETA

→

49

49.28

DIF. 0,28

PASADAS:
2

$0,28 \div 0,01 = 28$

$a \sin(x)$

$+ b x \sin(x)$

REPETIDO

Empujar y repetir

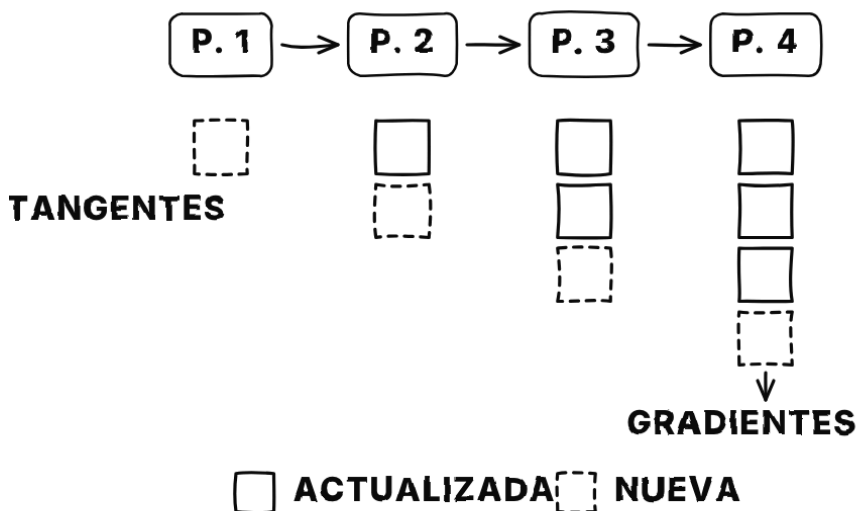
Una segunda idea es dejar que un ordenador haga álgebra de instituto. Dele a un programa de matemáticas simbólicas una fórmula hecha de un seno y unas cuantas multiplicaciones, y le devolverá la fórmula de su derivada, igual que haría un estudiante en papel. El

libro muestra un breve listado de Python que hace exactamente esto con la biblioteca SymPy e imprime la derivada como una fórmula nueva. El problema es que una fórmula escrita entera se repite: el mismo seno aparece en varios sitios, y un programa sensato lo calcularía una sola vez y guardaría el valor. Encontrar la mejor forma posible de compartir todas esas piezas repetidas resulta ser uno de esos problemas para los que no se conoce ningún método general rápido. La diferenciación automática esquiva esa búsqueda siguiendo la estructura que el programa ya tiene, puesto a puesto, y el libro añade que acaba siendo igual a un método simbólico al que se le permite guardar y reutilizar resultados intermedios.

Pruébelo

Escriba la receta: empiece con tres, duplíquelo, súmele uno, elévelo al cuadrado; y hágala dos veces en la calculadora de su teléfono, una desde tres y otra desde tres coma cero uno. Debería ver que el resultado se mueve unos cero coma veintiocho, veintiocho veces su empujón. Ahora imagine repetir esas dos ejecuciones para cada uno de los miles de millones de números de un modelo, y la necesidad de algo mejor resulta evidente.

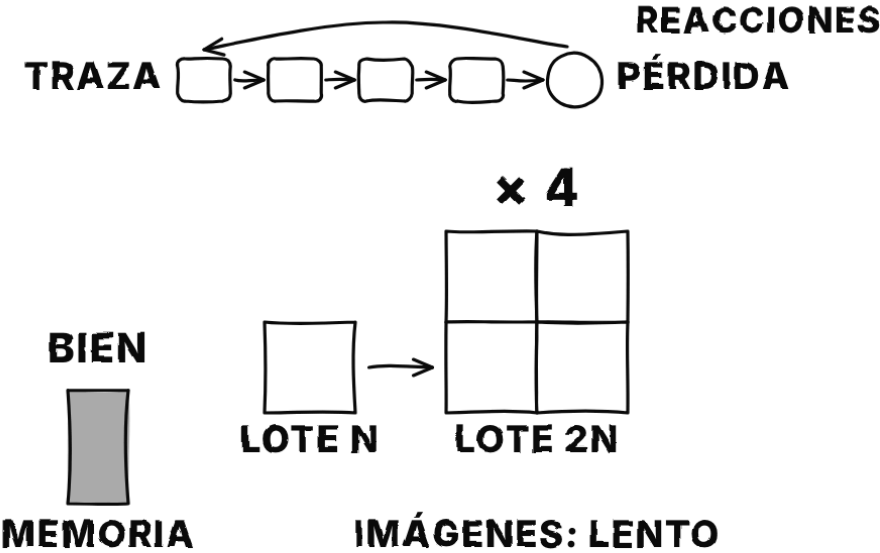
Así que el libro pasa al primer método de verdad, el modo directo, que se apoya en la regla de la cadena. Si un puesto alimenta a otro, la reacción de la pareja es la reacción del segundo multiplicada por la reacción del primero. Léalo como contabilidad a lo largo de la cadena: una vez que sabe cómo reacciona la salida del primer puesto a sus parámetros, y el segundo puesto se ejecuta, actualiza ese conocimiento multiplicándolo por el jacobiano de entrada del segundo puesto.



El modo directo guarda tablas

El modo directo guarda una tabla en curso para cada conjunto de parámetros encontrado hasta ahora, llamada tangente. En el primer puesto empieza una tabla para los primeros parámetros, que es simplemente el jacobiano de los pesos de ese puesto. En el segundo puesto hace dos cosas: actualiza la primera tabla con el jacobiano de entrada del segundo puesto, y empieza una tabla nueva para los segundos parámetros. En cada puesto siguiente se repite el mismo patrón, así que el décimo puesto debe actualizar nueve tablas guardadas y empezar una décima, todas con el mismo jacobiano de entrada de ese puesto. Cuando se llega al paso final de la suma, cada tabla se reduce al gradiente de sus parámetros, y el trabajo está hecho.

La buena noticia es la memoria: toda esta contabilidad avanza junto con el programa, así que necesita más o menos tanta memoria como el propio programa. La mala noticia es el tiempo. Cada actualización multiplica dos grandes tablas de reacciones, y cuando una capa trabaja con un lote de ejemplos, el coste crece con el cuadrado del tamaño del lote y con el cuadrado del número de valores por ejemplo. Duplique el lote de fotos con que se entrena una aplicación de fotos, y esa parte del trabajo se multiplica por cuatro; duplique el tamaño de la descripción de cada imagen, y vuelve a multiplicarse por cuatro. Para entradas de muchas dimensiones, como las imágenes, el modo directo se vuelve enseguida imposible, y ese es el problema que resolverá el camino de vuelta.



Memoria bien, tiempo al cuadrado

Para recordar

Un programa de entrenamiento es una cadena de puestos, la traza de evaluación, que termina sumando las penalizaciones de un lote en un único número, la pérdida. La diferenciación automática consiste en hallar cómo reacciona ese único número a los parámetros de cada puesto, de forma eficiente y todo a la vez, a partir de las reacciones propias de cada puesto. Empujar cada número y volver a ejecutar el modelo cuesta dos ejecuciones por número, y escribir las fórmulas de las derivadas repite trabajo, así que ninguno de los dos sirve para entrenar. El modo directo sigue la cadena, guardando y actualizando una tabla para cada conjunto de parámetros encontrado hasta el momento; ahorra memoria, pero cada actualización multiplica tablas enormes. Ese coste crece con el cuadrado del tamaño del lote y de los valores por ejemplo, lo que descarta el modo directo para entradas grandes como las imágenes.

Preguntas

1. ¿Por qué un programa de entrenamiento termina siempre en un único número?
 - a. Porque el modelo solo puede responder a un ejemplo cada vez
 - b. Porque se suman las penalizaciones del lote, y un número es lo que un optimizador puede empujar cuesta abajo

- c. Porque el primer puesto de la cadena reduce cada ejemplo a un número
 - d. Porque la regla de la cadena no puede tratar más de una salida
2. ¿Por qué empujar cada número y volver a ejecutar el modelo no sirve para entrenar un gran asistente de chat?
- a. Porque las respuestas que da siempre son erróneas
 - b. Porque solo funciona para números menores que uno
 - c. Porque necesita unas dos ejecuciones del modelo entero por cada parámetro, miles de millones de ejecuciones por paso de entrenamiento
 - d. Porque necesita escribir antes la fórmula de la derivada
3. En el modo directo, ¿qué debe hacer el décimo puesto de la cadena?
- a. Actualizar las nueve tablas guardadas hasta entonces y empezar una décima
 - b. Esperar a conocer el número final y luego volver hacia atrás
 - c. Solo empezar su propia tabla y dejar en paz las anteriores
 - d. Volver a ejecutar los nueve primeros puestos con números empujados
4. Si una aplicación de fotos duplica el tamaño de su lote de entrenamiento, ¿qué le pasa al coste de las actualizaciones de tablas del modo directo?
- a. Se queda igual, porque lo que crece es la memoria
 - b. Se duplica, igual que el lote

- c. Se multiplica más o menos por cuatro, porque crece con el cuadrado del tamaño del lote

Respuestas

1. **b)** El último puesto de verdad pone a cada ejemplo una penalización y un paso final las suma; un número es lo que un optimizador puede empujar cuesta abajo.
2. **c)** La diferenciación numérica cuesta dos ejecuciones por número, así que un modelo con miles de millones de parámetros necesitaría miles de millones de ejecuciones por paso; solo se conserva para comprobar otros métodos.
3. **a)** En cada puesto, el modo directo actualiza todas las tablas guardadas con el jacobiano de entrada de ese puesto y empieza una tabla nueva para los parámetros propios del puesto.
4. **c)** Cada actualización multiplica dos grandes tablas, y el coste crece con el cuadrado del tamaño del lote y de los valores por ejemplo.

Ejemplos resueltos

Medir una reacción empujando La receta es: empiece con 3, duplíquelo, súmele 1, elévelo al cuadrado. Hágala desde 3 y desde 3,01 y use los dos resultados para estimar con qué fuerza reacciona el resultado al número inicial.

Desde 3: duplicado es 6, más 1 es 7, al cuadrado es 49.

Desde 3,01: duplicado es 6,02, más 1 es 7,02, al cuadrado es 49,2804.

El resultado se movió 0,2804 mientras el inicio se movió 0,01.

Dividir lo que se movió el resultado entre el empujón da la reacción: 0,2804 dividido entre 0,01.

Respuesta: Unos 28; hicieron falta dos ejecuciones completas para conocer la reacción a un solo número.

Ejercicios

1. Un modelo pequeño tiene 1000 parámetros. Empujando y repitiendo en su forma más simple, dos ejecuciones por número, ¿cuántas ejecuciones del modelo necesita un paso de entrenamiento?

Pista: Cada número que hay que derivar necesita su propio par de ejecuciones.

Respuesta: 2000 ejecuciones para un paso, en lugar de unas pocas; por eso el libro reserva este método solo para comprobar otro más ingenioso.

2. Explique con sus propias palabras por qué el modo directo ahorra memoria pero es demasiado lento para los modelos de fotos.

Pista: Pregúntese qué tiene que guardar mientras avanza y qué tiene que multiplicar en cada puesto.

Respuesta: Su contabilidad avanza junto con el programa, así que necesita más o menos tanta memoria como el propio programa. Pero en cada puesto multiplica grandes tablas de reacciones para cada conjunto de parámetros encontrado hasta entonces, y ese coste crece con el cuadrado del tamaño del lote y de los valores por ejemplo, lo que es imposible para imágenes.

3. Un programa de matemáticas simbólicas escribe la derivada de una fórmula en la que el mismo seno aparece varias veces. ¿Por qué esa todavía no es una buena forma de derivar un modelo?

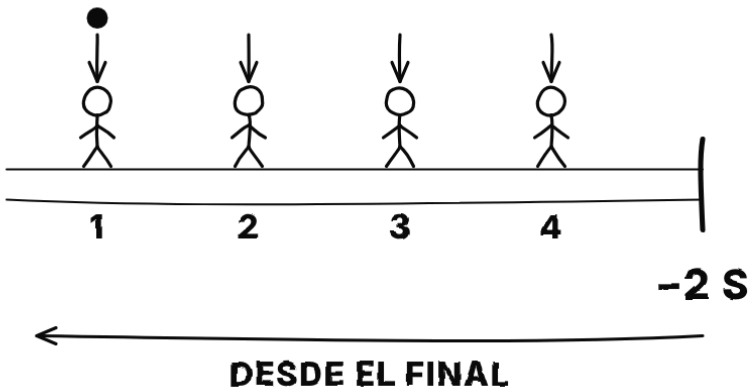
Pista: Piense en qué hace una fórmula escrita entera con las piezas que se repiten.

Respuesta: La fórmula escrita entera repite las mismas piezas, mientras que un programa sensato calcularía cada una una vez y la reutilizaría; encontrar la mejor forma de compartir todas las piezas repetidas es un problema sin ningún método general rápido conocido. La diferenciación automática evita esa búsqueda siguiendo la estructura que el programa ya tiene.

Volver desde la puntuación

Piense en una carrera de relevos que su equipo acaba de perder por dos segundos. La entrenadora no pide a cada corredor que repita el recorrido con unas zapatillas un poco distintas. Empieza en la línea de meta, donde hay un único número, los dos segundos, y vuelve por la pista preguntando a cada corredor, uno tras otro, cuánto de ese número pasó por su tramo. Cada respuesta es corta, y cada una prepara la siguiente pregunta para el corredor anterior. Ese camino de vuelta desde un único número final es toda la idea de esta parte.

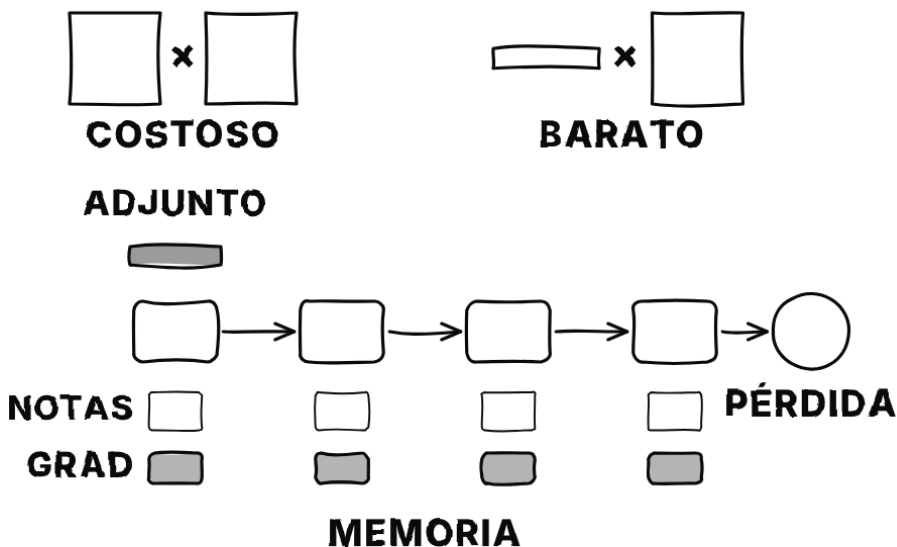
ENTRENADORA



Volver desde la meta

El modo directo, como vimos, llevaba una gran tabla para cada conjunto de parámetros a lo largo de la cadena y multiplicaba tablas enormes en cada puesto. El libro mira ahora la cadena de reacciones para un conjunto de parámetros, escrita entera. Es una larga fila de multiplicaciones: en su extremo está el paso final de la suma, después los jacobianos de entrada de todos los puestos posteriores, uno tras otro, y por último el jacobiano de los pesos del puesto que nos interesa. La multiplicación de tablas así puede hacerse en cualquier orden y da el mismo resultado. El modo directo simplemente recorría la fila en el orden en que se ejecutaba el programa; el truco está en empezar por el otro extremo.

Empezar por el final trae dos regalos. El primero es que el final es diminuto: como el programa termina en un único número, el último término no es una gran tabla sino una simple fila de valores, y multiplicar una fila por una tabla es mucho más barato que multiplicar dos tablas, y da otra fila. El segundo es que el largo tramo de jacobianos de entrada, desde el final hasta cualquier puesto, puede construirse paso a paso, añadiendo cada vez un puesto más en el lado izquierdo del camino. Así que una pequeña fila de valores viaja hacia atrás, puesto a puesto, y en cada parada se reutiliza en lugar de volver a calcularse.

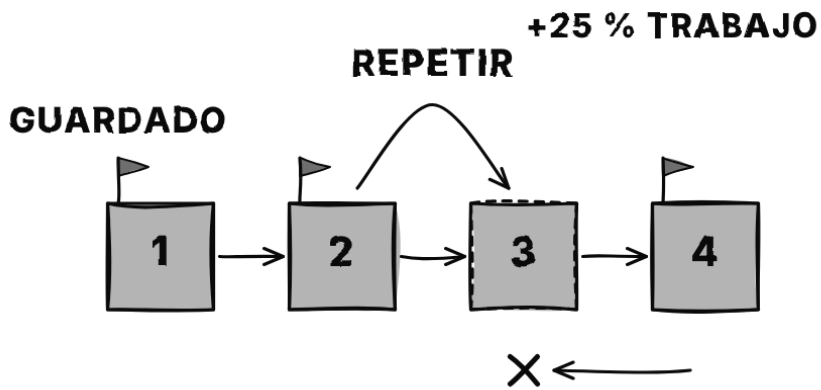


Una fila pequeña viaja hacia atrás

Todo junto, esto es la diferenciación automática en modo inverso. Primero se ejecuta todo el programa con normalidad, y se guarda cada resultado intermedio. Luego se coloca en la meta una fila de unos, que representa la suma final. Volviendo hacia atrás, en cada puesto ocurren dos cosas baratas: la fila viajera se combina con el jacobiano de los pesos de ese puesto, lo que da exactamente el gradiente de sus parámetros, y luego se combina con el jacobiano de entrada del puesto, lista para el puesto anterior. Las filas viajeras se llaman adjuntos; cada una dice cómo reacciona el número final a la salida de un puesto. El camino de vuelta es un reflejo del programa original, y en la práctica a las dos mitades se las llama pasada hacia delante y pasada hacia atrás, que no hay que confundir con el modo directo.

Esto es lo que ocurre cada vez que un asistente de chat aprende de un lote: una pasada hacia delante para obtener la pérdida, una pasada hacia atrás para obtener todos los gradientes. Cada paso del camino cuesta solo en proporción a los tamaños en juego, nunca a sus cuadrados, así que el modo inverso es muchísimo más rápido que el directo. El precio es la memoria, porque el camino de vuelta necesita los resultados intermedios de cada puesto, y hay que guardarlos todos hasta que llegue a ellos.

El libro muestra una forma de cambiar parte de esa memoria por tiempo, llamada puntos de control del gradiente. Imagine cuatro bloques en fila. En la pasada hacia delante solo se guardan las salidas del primer, segundo y cuarto bloque; esos son los puntos de control. El camino de vuelta va bien hasta llegar al tercer bloque, cuyos resultados se tiraron, así que se detiene. El programa vuelve entonces a ejecutar la pasada hacia delante desde el punto de control más cercano, el segundo bloque, para regenerar lo que faltaba, y el camino continúa. En este ejemplo, el trabajo completo cuesta una cuarta parte más de cálculo que una pasada hacia atrás normal, y en general, cuantos menos puntos de control se guardan, más recálculo hay que pagar.



Los puntos de control ahorran memoria

Para recordar

La cadena de reacciones para cualquier conjunto de parámetros es una larga fila de multiplicaciones que pueden hacerse en cualquier orden. Como el programa termina en un número, empezar por el final significa llevar hacia atrás una pequeña fila de valores en lugar de multiplicar grandes tablas, y cada paso reutiliza el anterior. El modo inverso ejecuta el programa una vez, guarda cada resultado intermedio y luego vuelve hacia atrás, produciendo todos los gradientes con un coste modesto; esto es la pasada hacia delante y la pasada hacia atrás de todo modelo entrenado. Su punto débil es la memoria, ya que todos los resultados intermedios deben esperar al camino de vuelta. Los puntos de control del gradiente guardan solo algunos resultados y recalculan el resto durante el camino de vuelta, cambiando cálculo extra por menos memoria.

Preguntas

1. ¿Por qué es más barato recorrer la cadena de reacciones empezando por el número final?
 - a. Porque el número final es pequeño, así que las tablas cercanas al final están vacías
 - b. Porque el programa termina en un número, así que el final es una fila pequeña, y una fila por una tabla es mucho más barato que una tabla por una tabla
 - c. Porque los puestos cercanos al final no tienen parámetros

- d. Porque volver hacia atrás no necesita resultados intermedios
2. ¿Qué ocurre en cada puesto durante el camino de vuelta?
- a. El puesto se vuelve a ejecutar con los parámetros empujados
 - b. La fila viajera se combina con el jacobiano de los pesos para dar el gradiente de ese puesto, y luego con el jacobiano de entrada para el puesto anterior
 - c. Se empieza una tabla nueva para cada puesto anterior
 - d. El resultado intermedio del puesto se tira y el camino continúa
3. ¿Cuál es el principal precio del modo inverso frente al modo directo?
- a. Necesita mucho más tiempo en cada paso de entrenamiento
 - b. Solo da gradientes aproximados
 - c. Necesita mucha memoria, porque cada resultado intermedio debe guardarse hasta que llegue a él el camino de vuelta
 - d. Solo puede usarse en modelos de cuatro bloques
4. Si un entrenamiento guarda menos puntos de control, ¿qué ocurre?
- a. Usa menos memoria pero tiene que recalcular más durante la pasada hacia atrás
 - b. Usa más memoria y calcula menos
 - c. No cambia nada, ya que los puntos de control solo afectan a la pasada hacia delante
 - d. Los gradientes se vuelven menos precisos

Respuestas

1. **b)** El programa termina en un único número, así que el último término es una simple fila de valores; multiplicar una fila por una tabla es mucho más barato que multiplicar dos tablas, y da otra fila.
2. **b)** En cada puesto, la fila viajera da el gradiente de sus parámetros a través del jacobiano de los pesos y luego se pasa a través del jacobiano de entrada, lista para el puesto anterior.
3. **c)** El modo inverso es mucho más rápido, pero el camino de vuelta necesita los resultados intermedios de cada puesto, así que hay que guardarlos todos hasta que llegue a ellos.
4. **a)** Los puntos de control cambian memoria por tiempo: cuantos menos resultados se guardan, más hay que recalcular desde el punto de control más cercano durante el camino de vuelta.

Ejemplos resueltos

Puntos de control en cuatro bloques Un modelo tiene cuatro bloques del mismo coste. En la pasada hacia delante solo se guardan las salidas de los bloques uno, dos y cuatro. ¿Cuánto cálculo extra necesita la pasada hacia atrás, en comparación con guardarlo todo?

El camino de vuelta pasa el bloque cuatro y llega al bloque tres, cuyos resultados no se guardaron.

Vuelve a ejecutar la pasada hacia delante desde el punto de control más cercano, el bloque dos, para regenerar los resultados del bloque tres.

Esa repetición cuesta el trabajo hacia delante de un bloque, además de los cuatro bloques de la pasada hacia delante normal.

Un bloque extra de cuatro es una cuarta parte más de cálculo.

Respuesta: Unas 1,25 veces el cálculo, una cuarta parte más, a cambio de no guardar los resultados del bloque tres.

Ejercicios

1. Explique con sus propias palabras por qué se puede cambiar el orden de las multiplicaciones en la cadena de reacciones.

Pista: Piense en qué tipo de operación forma la cadena.

Respuesta: La cadena no es más que una larga fila de multiplicaciones de tablas, y esas multiplicaciones dan el mismo resultado sea cual sea la pareja que se combine primero. Así que somos libres de empezar por el número final en lugar de por el primer puesto, y esa elección es lo que hace barato el modo inverso.

2. El modelo de fotos de un teléfono es demasiado grande para guardar en memoria todos los resultados intermedios durante el entrenamiento. ¿Qué pueden hacer sus ingenieros, y qué les cuesta?

Pista: Piense en guardar solo algunos resultados.

Respuesta: Pueden usar puntos de control del gradiente: guardar solo unos pocos resultados intermedios, los puntos de control, y recalcular los demás desde el punto de control más cercano durante la pasada hacia atrás. La memoria necesaria baja, y el precio es cálculo extra, que crece cuantos menos puntos de control guarden.

3. ¿Por qué el modo inverso debe ejecutar todo el programa antes de poder calcular ningún gradiente, mientras que el modo directo podía trabajar a la vez que el programa?

Pista: ¿Dónde empieza el camino de vuelta?

Respuesta: El camino de vuelta empieza en el número final, al terminar el programa, así que ese número, y cada resultado intermedio que necesitará el camino, deben existir antes. El modo directo sigue el orden propio del programa, así que su contabilidad puede avanzar junto con él.

Hacerle una sola pregunta a la tabla

Imagine un puesto de mercado con un enorme tablero de precios: para cada fruta, cuánto cuesta cada tipo de zumo en que se usa. Usted se acerca con una lista de la compra que dice cuántas botellas de cada zumo quiere. No necesita que el vendedor le copie el tablero entero; necesita una sola respuesta: cuánto le cuesta su lista por fruta. El vendedor puede calcularlo de cabeza, fila a fila, sin escribir nunca el tablero. Esa diferencia, entre entregar una tabla entera y responder a una pregunta que se le hace, es el truco de esta parte.

VJP



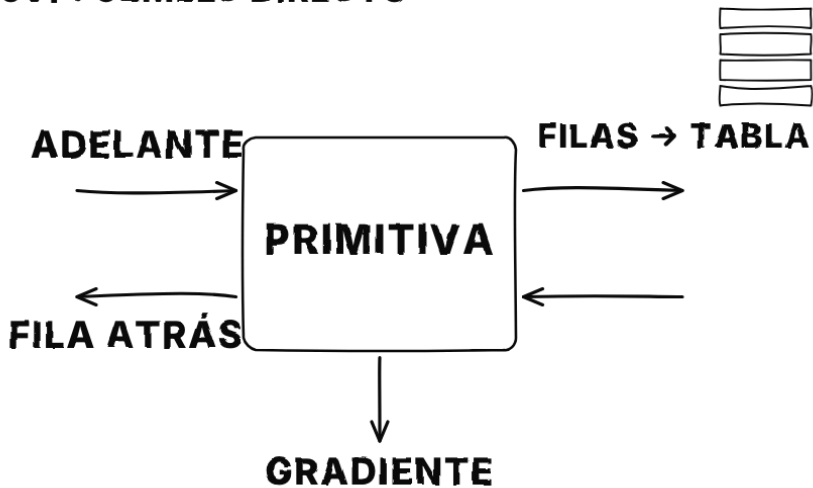
× 2 POR PUESTO

Una pregunta, no la tabla

Mire otra vez el camino de vuelta. En cada puesto solo ocurría un tipo de operación: la pequeña fila viajera se combinaba con una de las tablas del puesto, su jacobiano de entrada o su jacobiano de los pesos. El libro le da un nombre a esa operación, el producto vector-jacobiano, VJP para abreviar: entra una fila de «cuánto importa cada salida» y sale una fila de «cuánto importa cada entrada». Por tanto, cada puesto necesita exactamente dos de ellos, uno para su entrada y otro para sus parámetros, y una llamada a cada uno basta para llevar la pasada hacia atrás a través de ese puesto.

Aquí está la consecuencia, y es grande. Todo un sistema de diferenciación automática puede construirse eligiendo un conjunto de primitivas y dando a cada una sus dos VJP, sin escribir nunca un jacobiano completo en memoria. El libro lo dibuja como una caja por primitiva, con la puerta habitual hacia delante y dos puertas extra hacia atrás: una devuelve la fila viajera al puesto anterior y la otra entrega el gradiente de los parámetros de la caja. No se pierde nada: si alguna vez quisiera la tabla completa, podría reconstruirla fila a fila preguntando al VJP por una sola salida cada vez. Para su imagen en espejo, el modo directo, el libro señala que existe una operación gemela que va en sentido contrario, el producto jacobiano-vector, JVP para abreviar.

JVP: GEMELO DIRECTO

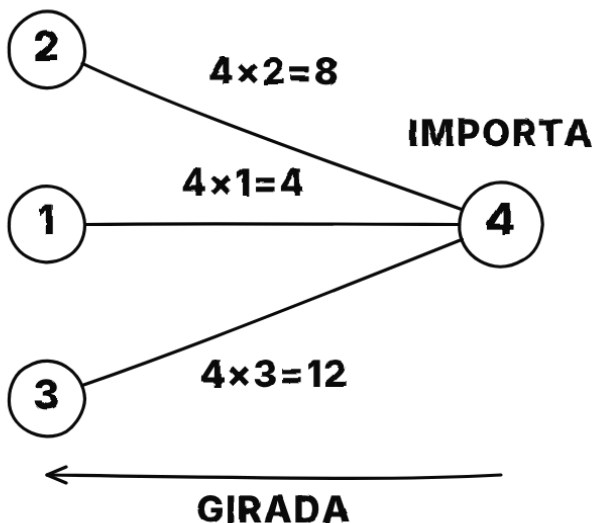


Una caja con dos puertas traseras

¿Por qué es esto tanto mejor? Tome el puesto más común de todos, una capa que mezcla sus entradas con una cuadrícula de pesos, como hace cada capa de un modelo de chat o de un modelo de fotos muchas veces. Su tabla de reacciones respecto de sus pesos ni siquiera es una tabla plana, sino un bloque de tres dimensiones, porque cada salida puede reaccionar a cada peso de la cuadrícula. Y sin embargo, la pregunta que le hace el camino de vuelta tiene una respuesta asombrosamente sencilla. Para devolver la fila viajera, la capa la pasa por su propia cuadrícula de pesos, girada de modo que salidas y entradas intercambian sus puestos. Y el gradiente de un solo peso es simplemente dos números multiplicados: cuánto importa la salida a la que alimenta, por el valor de entrada que multiplicó en el camino de ida.

ENTRADAS

SIN BLOQUE 3D



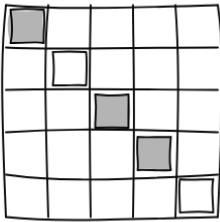
La parte de culpa de un peso

Ese último hecho merece decirse otra vez con otras palabras. Cada peso conecta una entrada con una salida, así que su parte de culpa es la importancia de su salida por el tamaño de su entrada, y toda la cuadrícula de estos productos se rellena de una vez. El bloque de tres dimensiones nunca tiene por qué existir.

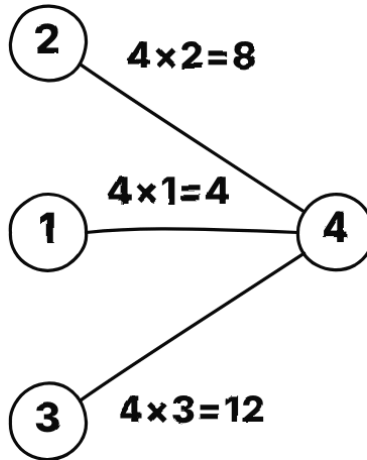
El segundo puesto común es una función de activación como la ReLU, que trata cada valor por separado y no tiene parámetros, así que solo hace falta un VJP. Su tabla está casi entera vacía, porque cada salida depende solo de su propia entrada; todo lo que importa está en la diagonal, la pendiente de la función en cada valor. Así que la pregunta de vuelta se responde escalando cada casilla de la fila viajera por esa pendiente, una a una. Para una ReLU, eso significa dejar pasar el

valor donde la entrada era positiva y detenerlo donde la entrada era negativa. De nuevo, la tabla completa, con su océano de ceros, nunca se construye.

TABLA RELU



3 0 2 4 0



SIN TABLA

La ReLU solo escala la fila

Pruébalo

Tome tres números como entradas de una capa, digamos dos, uno y tres, e imagine una salida que importa en una cantidad de cuatro. El gradiente del peso que une esa salida con la primera entrada es cuatro por dos; para la segunda, cuatro por uno; para la tercera, cuatro por tres. Ha rellenado una fila de gradientes con tres multiplicaciones y sin ninguna tabla.

Para recordar

El camino de vuelta solo multiplica una fila pequeña por una tabla del puesto, y esa operación se llama producto vector-jacobiano. Cada primitiva necesita dos, uno para su entrada y otro para sus parámetros, y solo con ellos puede construirse un sistema completo de diferenciación automática sin guardar ningún jacobiano. En una capa de pesos, la fila se devuelve a través de los pesos girados, y el gradiente de cada peso es la importancia de su salida por la entrada que multiplicó. En una activación como la ReLU, la fila simplemente se escala, valor a valor, por la pendiente de la función. En ambos casos las enormes tablas de reacciones nunca se escriben, y eso es lo que hace posible entrenar modelos grandes.

Preguntas

1. ¿Por qué un sistema de diferenciación automática puede funcionar sin guardar nunca un jacobiano completo?
 - a. Porque los jacobianos son siempre lo bastante pequeños como para ignorarlos
 - b. Porque el camino de vuelta solo necesita una fila por una tabla, y cada primitiva puede responder a eso directamente con sus dos VJP
 - c. Porque el jacobiano completo se calcula una vez y luego se tira
 - d. Porque solo el modo directo necesita jacobianos

2. En una capa que mezcla sus entradas con una cuadrícula de pesos, ¿cuál es el gradiente de un solo peso?
 - a. La importancia de la salida a la que alimenta por el valor de entrada que multiplicó
 - b. El valor del propio peso por el tamaño de la cuadrícula
 - c. La suma de todas las entradas de la capa
 - d. La pendiente de la función de activación en ese peso
3. ¿Por qué un puesto ReLU necesita un solo VJP en lugar de dos?
 - a. Porque no tiene parámetros, así que no hay gradiente de pesos que calcular
 - b. Porque su pendiente es siempre uno
 - c. Porque siempre es el último puesto del modelo
 - d. Porque de ella se encarga el modo directo
4. Si de verdad quisiera la tabla completa de un puesto, ¿qué podría hacer?
 - a. Nada; una vez que se usan los VJP, la tabla se pierde para siempre
 - b. Reconstruirla fila a fila preguntando al VJP por una sola salida cada vez
 - c. Ejecutar dos veces la pasada hacia delante y comparar
 - d. Multiplicar entre sí los VJP de todos los puestos

Respuestas

1. **b)** La única operación que necesita el camino de vuelta es el producto vector-jacobiano, así que basta con dar a cada primitiva sus dos VJP, y ningún jacobiano se escribe nunca en memoria.

2. a) Cada peso conecta una entrada con una salida, así que su parte de culpa es la importancia de su salida por el tamaño de su entrada.

3. a) Una activación como la ReLU trata cada valor por separado y no tiene parámetros, así que solo hace falta el VJP de su entrada.

4. b) No se pierde nada: preguntar al VJP por una sola salida cada vez da la tabla completa fila a fila.

Ejemplos resueltos

Una cuadrícula de gradientes de los pesos Una capa pequeña recibe tres entradas, 2, 1 y 3, y tiene dos salidas. La fila viajera dice que la primera salida importa 4 y la segunda 1. ¿Cuáles son los gradientes de los seis pesos?

Cada peso une una entrada con una salida, así que su gradiente es la importancia de esa salida por esa entrada.

Para la primera salida, que importa 4: 4 por 2, 4 por 1, 4 por 3, lo que da 8, 4 y 12.

Para la segunda salida, que importa 1: 1 por 2, 1 por 1, 1 por 3, lo que da 2, 1 y 3.

Juntos rellenan toda la cuadrícula de gradientes de los pesos sin construir ningún bloque de tres dimensiones.

Respuesta: Pesos de la primera salida: 8, 4, 12; pesos de la segunda salida: 2, 1, 3.

Ejercicios

1. Una ReLU recibió las entradas 1,5, menos 0,7 y 2 en la pasada hacia delante. En el camino de vuelta, la fila viajera que le llega es 5, menos 2 y 3. ¿Qué fila devuelve?

Pista: La pendiente de la ReLU es uno donde la entrada era positiva y cero donde era negativa.

Respuesta: 5, 0 y 3: el primer y el tercer valor pasan porque sus entradas eran positivas, y el segundo se detiene porque su entrada era negativa.

2. Explique con sus propias palabras por qué el gradiente de un peso debería ser la importancia de su salida por la entrada que multiplicó.

Pista: Piense en lo que hace un peso en la pasada hacia delante.

Respuesta: En la pasada hacia delante, un peso multiplica una entrada y suma el resultado a una salida. Subir el peso mueve por tanto esa salida en proporción a la entrada, y el efecto de la salida sobre el número final es su importancia; así que el efecto del peso es las dos cosas multiplicadas.

3. ¿Por qué se describe la tabla de una ReLU como un océano de ceros, y por qué importa eso?

Pista: ¿De qué depende cada salida de la ReLU?

Respuesta: Cada salida depende solo de su propia entrada, así que la tabla solo tiene valores en su diagonal y ceros en todo lo demás. Construir la desperdiciaría memoria en ceros, mientras que el VJP simplemente escala cada valor por la pendiente, sin necesitar ninguna tabla.

Cómo vuelven atrás los entornos, y por qué ganó la ReLU

Ponga una página en una fotocopidora, ajústela para reducir a una cuarta parte de su tamaño, copie la copia y vuelva a copiar esa copia. Tras dos rondas el texto es un borrón; tras diez, la página es una mota que podría tapar con un grano de arena. Cada ronda solo encogía las cosas por un factor fijo, pero los factores se multiplicaron. Tenga presente esa fotocopidora; está a punto de explicar por qué una pequeña función conquistó la IA moderna. Pero primero, el libro mira cómo hacen el camino de vuelta los entornos reales.



PYTORCH: TORCH.FUNC

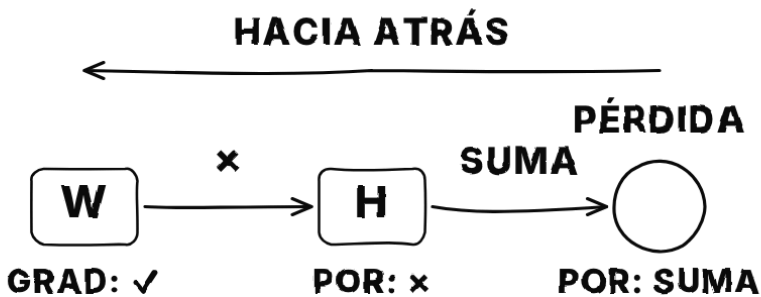


Una herramienta que devuelve una función

Hay muchas formas de construir el modo inverso en software, desde llevar una lista en curso de cada operación realizada, como hace TensorFlow, hasta reescribir el código del programa en su gemelo hacia atrás. En JAX, donde todo es una función, las propias herramientas son funciones que reciben funciones. Usted le da a JAX una función y recibe una función nueva que calcula su gradiente. Con el mismo espíritu, la herramienta VJP recibe una función y un punto y devuelve una pequeña función que responde a la pregunta hacia atrás en ese punto. El breve ejemplo del libro define una función que eleva al cuadrado diez números y los suma, pide su función gradiente y le da diez números al azar; salen diez números, una reacción por cada entrada. PyTorch ofrece el mismo estilo en un módulo propio, y combinando estas herramientas con astucia se puede obtener aún más, como las segundas derivadas.

Sin embargo, la mayoría de los modelos se escriben como objetos, capas que guardan dentro sus parámetros, como vimos al principio. Una forma de llevarlos al estilo de funciones es «purificarlos»: sacar los parámetros a una colección aparte y llamar al modelo como una función simple de su entrada y de esos parámetros. PyTorch tiene además su propia forma incorporada, que no necesita ese paso. Cada bloque de números que calcula lleva dos notas extra: un hueco vacío para su futuro gradiente y un registro de la operación que lo produjo. La figura del libro lo muestra: a la izquierda, los números con sus dos notas; a la derecha, el camino de vuelta siguiendo los registros desde la pérdida final hacia atrás por el grafo, dejando gradientes en los huecos a su paso. Para ahorrar memoria, solo se llenan los huecos de los números que han pedido un gradiente, normalmente los parámetros que se entrenan. Eso es literalmente lo que ocurre dentro del entrenamiento del modelo de una aplicación de fotos cuando el

programa pide el paso hacia atrás sobre la pérdida, y el autor recomienda con entusiasmo construir usted mismo una versión diminuta de un sistema así, como hacen muchos proyectos didácticos en internet.

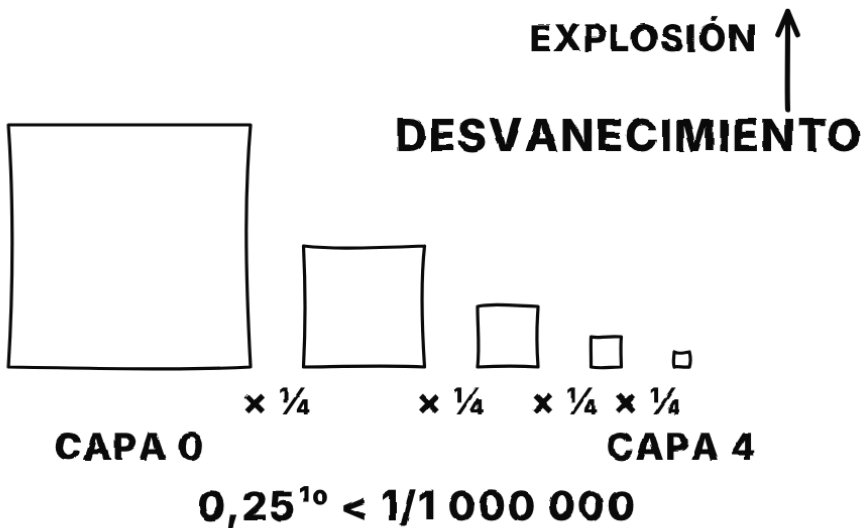


SOLO SI SE PIDE

Etiquetas seguidas hacia atrás

Volvamos ahora a la fotocopidora. Recuerde que, cuando el camino de vuelta pasa por una función de activación, la fila viajera se escala por la pendiente de la función. Un modelo profundo tiene muchos puestos así, de modo que la fila se escala una y otra vez, y los factores se multiplican exactamente como las reducciones de la fotocopidora. Si las pendientes están por debajo de uno en todas partes, el gradiente se encoge hacia nada a un ritmo que crece con cada capa, lo que se llama el problema del desvanecimiento del gradiente. Si están por

encima de uno en todas partes, crece sin límite: el problema de la explosión del gradiente. Los dos son peligros reales porque los ordenadores guardan los números con una precisión limitada, normalmente treinta y dos cifras binarias o menos, así que un número que se hace demasiado pequeño se vuelve cero y uno que se hace demasiado grande desborda.

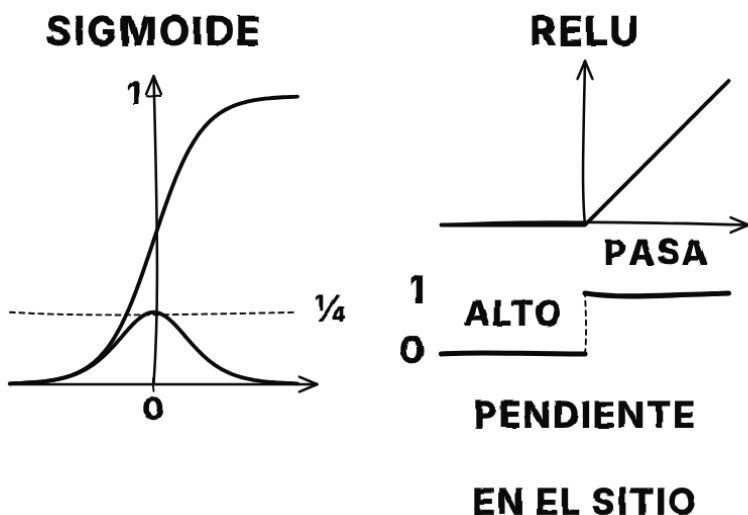


Factores que se multiplican

Pruébalo

En la calculadora de su teléfono, escriba cero coma veinticinco y multiplíquelo por cero coma veinticinco diez veces. Debería acabar con menos de una millonésima. Eso es más o menos lo que diez capas de la vieja activación favorita pueden hacerle a un gradiente en el mejor de los casos, y las redes reales tienen muchas más capas.

Esa vieja favorita es la sigmoide, una suave curva en forma de S que fue popular porque es una versión suave de un interruptor de encendido y apagado. Su pendiente es una joroba baja que nunca sube por encima de un cuarto, más alta en el centro de la S y casi plana en los dos extremos. Así que cada capa sigmoide encoge la fila de vuelta a un cuarto o menos, la fotocopidora en su ajuste más pequeño.



Pendiente de la sigmoide frente a pendiente de la ReLU

Lo ideal sería una función con pendiente uno en todas partes, pero la única función así es la que no cambia nada, y apilar capas que no cambian nada haría que todo el modelo no fuera más potente que una sola capa. Así que necesitamos una función lo bastante recta para mantener sanos los gradientes, pero lo bastante doblada para mantener separadas las capas. La ReLU encaja: es plana en cero para entradas negativas y una línea recta para las positivas, así que su pendiente es o cero o exactamente uno. Por eso, en el camino de vuelta cada valor o bien se detiene, lo que hace disperso el cálculo, o bien pasa intacto, sin encogerse ni dispararse. La figura del libro muestra la ReLU, con forma de bisagra, junto a su pendiente, un escalón que salta de cero a uno.

En un recuadro aparte, el autor añade una curiosidad: una pila de capas puramente lineales, ejecutada en un ordenador real, no es exactamente lineal, por pequeñas asperezas en el límite de la precisión de la máquina, y esto incluso se ha aprovechado para entrenar redes profundas totalmente lineales.

La ReLU tiene un pequeño regalo más. Su pendiente puede leerse en su salida igual de bien que en su entrada, porque solo borra los valores negativos y deja los positivos como están. Así que un entorno puede sobrescribir la entrada de la ReLU con su salida durante la pasada hacia delante y aun así hacer bien el camino de vuelta, ahorrando un poco de esa preciada memoria; PyTorch tiene una opción «en el sitio» exactamente para esto.

Para recordar

JAX trata los gradientes y los VJP como herramientas que reciben una función y devuelven una función nueva, mientras que PyTorch etiqueta cada bloque de números calculado con la operación que lo produjo y un hueco para su gradiente, y luego recorre esos registros hacia atrás desde la pérdida. Cada activación escala la fila de vuelta por su pendiente, y a lo largo de muchas capas esos factores se multiplican. Las pendientes por debajo de uno hacen que los gradientes se desvanezcan y las de por encima de uno hacen que exploten, algo que la precisión limitada del ordenador convierte en fallos reales. La pendiente de la sigmoide es como mucho un cuarto, así que pilas profundas de ella dejan sin gradiente a la red; la pendiente de la ReLU es o cero o uno, así que los valores o se detienen o pasan intactos. Como la pendiente de la ReLU puede leerse en su salida, su entrada puede sobrescribirse en el sitio para ahorrar memoria.

Preguntas

1. Cuando PyTorch ejecuta el paso hacia atrás sobre la pérdida, ¿cómo encuentra el camino de vuelta por el modelo?
 - a. Vuelve a ejecutar cada capa con números empujados
 - b. Sigue el registro que cada bloque de números guarda de la operación que lo produjo, dejando gradientes en los huecos a su paso
 - c. Le pide a JAX que calcule la función gradiente

- d. Escribe primero la fórmula de la derivada de todo el modelo
2. ¿Por qué las pilas profundas de capas sigmoides tienden a dejar sin gradiente a la red?
- a. Porque la sigmoide no tiene ninguna pendiente
 - b. Porque su pendiente nunca pasa de un cuarto, y las pendientes de todas las capas se multiplican en el camino de vuelta
 - c. Porque la sigmoide sobrescribe su entrada en el sitio
 - d. Porque las capas sigmoides tienen demasiados parámetros
3. ¿Por qué no usar simplemente una activación cuya pendiente sea uno en todas partes?
- a. Porque una función así no cambia nada, y las capas apiladas no serían más potentes que una sola
 - b. Porque una pendiente de uno hace explotar los gradientes
 - c. Porque los ordenadores no pueden guardar el número uno
 - d. Porque necesitaría un hueco de gradiente para cada valor
4. ¿Por qué un entorno puede sobrescribir la entrada de una ReLU con su salida durante la pasada hacia delante?
- a. Porque la pendiente de la ReLU puede leerse en su salida igual de bien que en su entrada
 - b. Porque la ReLU nunca aparece en el camino de vuelta
 - c. Porque ninguna capa posterior necesita ya la entrada
 - d. Porque sobrescribir hace más grande el gradiente

Respuestas

1. **b)** Cada bloque de números lleva un registro de la operación que lo produjo y un hueco para su gradiente; el camino de vuelta sigue esos registros desde la pérdida y llena los huecos.
2. **b)** Cada activación escala la fila de vuelta por su pendiente; la pendiente de la sigmoide es como mucho un cuarto, así que a lo largo de muchas capas el gradiente se encoge hasta casi nada.
3. **a)** La única función con pendiente uno en todas partes es la que no cambia nada; la activación debe estar lo bastante doblada para mantener separadas las capas.
4. **a)** La ReLU solo borra los valores negativos y deja los positivos sin cambios, así que su pendiente puede leerse en la salida y la entrada puede sobrescribirse para ahorrar memoria.

Ejemplos resueltos

El gradiente de una suma de cuadrados Como en el ejemplo del libro, una función eleva al cuadrado sus entradas y las suma. Para las tres entradas 1, menos 2 y 3, ¿qué devuelve su función gradiente?

El gradiente tiene una casilla por entrada: cuánto reacciona la suma a un empujón de esa entrada.

Cada entrada aparece solo en su propio cuadrado, así que para ella solo importa ese cuadrado.

Un cuadrado reacciona a un empujón pequeño con el doble del valor que se eleva al cuadrado.

Así que cada casilla es el doble de la entrada: 2 por 1, 2 por menos 2, 2 por 3.

Respuesta: 2, menos 4 y 6: tantos números como entradas había, igual que las diez entradas del libro dieron diez números.

Ejercicios

1. Un modelo tiene cinco capas sigmoides, y en el mejor de los casos cada una escala la fila de vuelta por un cuarto. ¿Cuánto se reduce, como mucho, el gradiente después de pasar por las cinco?

Pista: Los factores se multiplican: un cuarto por un cuarto, cinco veces.

Respuesta: Un cuarto elevado a cinco es 1 entre 1024, así que el gradiente se reduce, en el mejor caso, unas mil veces.

2. Sustituya ahora las cinco sigmoides por cinco ReLU, y suponga que el valor en cuestión es positivo en todas las capas. ¿Qué le pasa en el camino de vuelta?

Pista: ¿Cuál es la pendiente de la ReLU para entradas positivas?

Respuesta: La pendiente es exactamente uno en cada capa, así que el valor vuelve intacto, sin encogerse ni dispararse. Si hubiera sido negativo en alguna capa, se habría detenido allí.

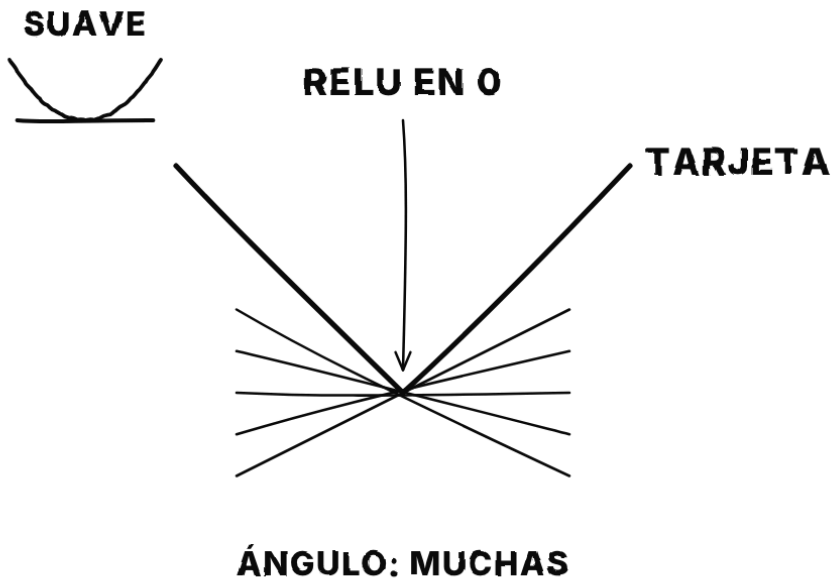
3. Explique con sus propias palabras por qué el desvanecimiento y la explosión del gradiente son un problema real en un ordenador y no solo en el papel.

Pista: Piense en cuántas cifras guarda un ordenador de cada número.

Respuesta: Los ordenadores guardan los números con una precisión limitada, normalmente 32 cifras binarias o menos. Un gradiente multiplicado por muchas pendientes pequeñas acaba siendo demasiado pequeño para guardarlo y se convierte en cero, mientras que uno multiplicado por muchas pendientes grandes se vuelve demasiado grande y desborda, así que el entrenamiento se rompe.

La esquina de la ReLU

Doble una postal en forma de V poco profunda y póngala de pie sobre la mesa, con la punta hacia abajo, como la gráfica de una ReLU con su esquina abajo. Ahora tome una regla e intente apoyarla en la esquina desde abajo, de modo que toque la esquina y quede por debajo de toda la tarjeta. En una curva suave habría exactamente una forma de hacerlo, la tangente. En la esquina, la regla puede inclinarse a cualquier ángulo entre las pendientes de los dos lados, y todas esas inclinaciones sirven. Esa pequeña observación es la forma en que el libro resuelve el último cabo suelto del capítulo.



Una regla bajo una tarjeta doblada

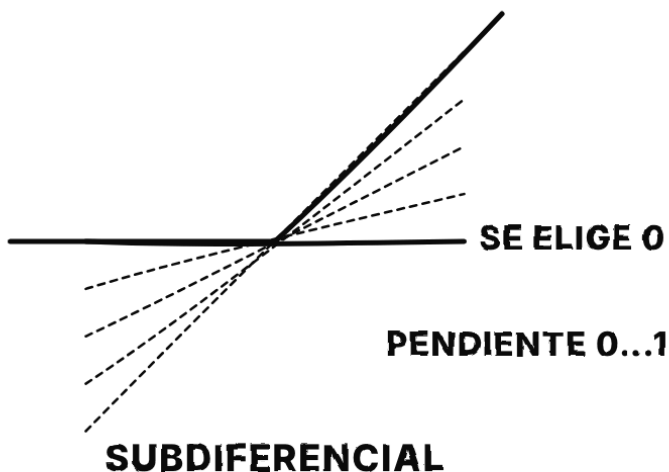
La ReLU, plana para los negativos y una recta para los positivos, tiene en cero una esquina donde su pendiente no está definida. ¿Qué hace allí el recorrido hacia atrás? La respuesta práctica es que apenas importa: el entrenamiento empieza con números aleatorios y avanza en pequeños pasos aleatorios, así que la probabilidad de caer exactamente en cero es prácticamente nula, y en todos los demás puntos la pendiente está perfectamente clara.

La respuesta cuidadosa usa la idea de la regla. Para una función que se curva hacia arriba como un cuenco, cualquier recta que la toque en un punto y quede por debajo de toda la función es allí un subgradiente, y su pendiente es un sustituto válido de la pendiente que falta. Donde la función es suave solo hay una recta así, la tangente de siempre; en una esquina hay muchas, y juntas forman el subdiferencial. Para la ReLU en cero, sirve cualquier pendiente de cero a uno, y la mayoría de los marcos de trabajo simplemente eligen cero. El entrenamiento que elige una pendiente así en cada paso se llama descenso por subgradiente.

RELU $\rightarrow 0$

RELU₂ $\rightarrow 1$

RELU₃ $\rightarrow 0,5$

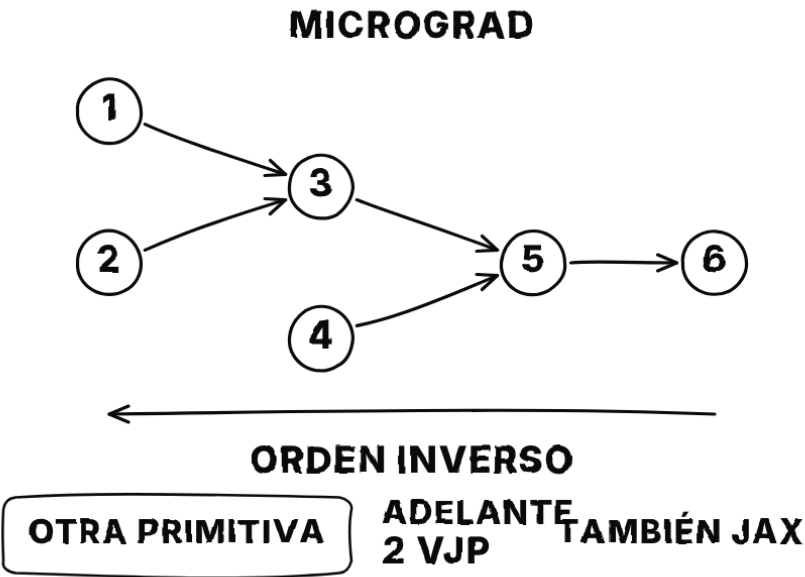


Cualquier pendiente de 0 a 1

El libro reconoce que la realidad es más complicada. Para funciones que no tienen forma de cuenco, la definición debe relajarse para que valga solo cerca del punto, y los sistemas de diferenciación automática pueden dar respuestas distintas según cómo se escriba una función. El ejemplo del autor es llamativo: tres maneras de escribir exactamente la misma ReLU hacen que PyTorch dé, en cero, una pendiente de cero, de uno y de un medio.

Para terminar, el libro pasa a la práctica. Su consejo más cálido es que usted mismo construya un pequeño sistema en modo inverso, como hace el proyecto didáctico *micrograd* para números sueltos. El único cuidado extra cuando un modelo se ramifica es poner las estaciones en el orden adecuado antes del recorrido hacia atrás, para que ningún

gradiente baje por un camino equivocado; micrograd lo hace con un paso de ordenación barato. Un segundo buen ejercicio es escribir una primitiva nueva en PyTorch, dándole una pasada hacia delante y sus VJP, como una de las activaciones entrenables del capítulo anterior, aunque PyTorch podría deducir la pasada hacia atrás por sí mismo. Lo mismo puede hacerse en JAX, que ofrece una versión didáctica de sí mismo, una forma de escribir VJP propios y un recetario de usos avanzados como las segundas derivadas. Y con esto, el recorrido hacia atrás que entrena a cada asistente de chat y a cada aplicación de fotos que usted usa ya no es un misterio, sino un método que usted mismo podría escribir.



Construya su propio recorrido hacia atrás

Para recordar

La ReLU tiene una esquina en cero, pero el entrenamiento casi nunca cae exactamente ahí, así que en la práctica su pendiente siempre está definida. Formalmente, un subgradiente es la pendiente de una recta que toca una función en forma de cuenco en un punto y queda por debajo de ella en todas partes; en una esquina existen muchas pendientes así, que forman el subdiferencial. Para la ReLU en cero cualquier pendiente de cero a uno es válida, y los marcos de trabajo suelen tomar cero. Distintas maneras de escribir la misma función pueden dar pendientes distintas en una esquina, como muestran el cero, el uno y el un medio de PyTorch. La mejor manera de hacer suyo el capítulo es construir un sistema diminuto en modo inverso o una primitiva propia con sus VJP.

Preguntas

1. ¿Por qué la pendiente no definida de la ReLU en cero apenas importa en el entrenamiento de cada día?
 - a. Porque los marcos de trabajo eliminan todos los ceros de los datos
 - b. Porque el entrenamiento empieza con números aleatorios y avanza en pequeños pasos aleatorios, así que caer exactamente en cero es prácticamente imposible
 - c. Porque la pendiente de la ReLU en cero es siempre uno

- d. Porque el recorrido hacia atrás se salta todas las estaciones ReLU
2. En una esquina de una función en forma de cuenco, ¿por qué hay muchos subgradientes en lugar de uno?
 - a. Porque muchas rectas pueden tocar la esquina sin dejar de estar por debajo de toda la función
 - b. Porque la función tiene muchas esquinas juntas
 - c. Porque en una esquina la función no tiene forma de cuenco
 - d. Porque cualquier recta que pase por la esquina es un subgradiente, sea cual sea su inclinación
 3. ¿Qué pendientes son subgradientes válidos de la ReLU en cero?
 - a. Solo cero
 - b. Solo uno
 - c. Cualquier pendiente de cero a uno
 - d. Cualquier pendiente
 4. ¿Qué muestra el ejemplo de PyTorch del libro sobre las esquinas?
 - a. Que PyTorch se niega a calcular gradientes en las esquinas
 - b. Que tres maneras de escribir la misma ReLU dan tres pendientes distintas en cero
 - c. Que la ReLU no es realmente igual a sus versiones reescritas
 - d. Que solo JAX trata bien las esquinas

Respuestas

1. **b)** La probabilidad de caer exactamente en cero es prácticamente nula, y en todos los demás puntos la pendiente de la ReLU está perfectamente

clara.

2. a) Un subgradiente es la pendiente de una recta que toca la función en el punto y queda por debajo de ella; en un punto suave solo lo consigue la tangente, en una esquina lo consigue toda una gama de inclinaciones.

3. c) Para la ReLU en cero sirve cualquier pendiente de cero a uno, y la mayoría de los marcos de trabajo simplemente eligen cero.

4. b) Tres maneras de escribir exactamente la misma ReLU hacen que PyTorch dé en cero una pendiente de cero, de uno y de un medio.

Ejemplos resueltos

Tres ReLU, tres pendientes Una segunda versión de la ReLU se escribe así: la ReLU de menos la entrada, más la entrada. Una tercera toma la mitad de la ReLU original más la mitad de la segunda. Compruebe que ambas son iguales a la ReLU y halle la pendiente que PyTorch da en cero para cada una, sabiendo que toma como 0 la pendiente de la propia ReLU en cero.

Para una entrada positiva, la ReLU de menos la entrada es cero, así que la segunda versión da la propia entrada, como la ReLU; para una entrada negativa, da menos la entrada más la entrada, que es cero, de nuevo como la ReLU.

La tercera versión promedia dos funciones que son ambas iguales a la ReLU, así que también es igual a la ReLU.

En cero, la pendiente de la segunda versión se toma como menos la pendiente de la ReLU, que es 0, más 1 de la entrada sumada, así que 1.

La pendiente de la tercera versión es la mitad de 0 más la mitad de 1, así que 0,5.

Respuesta: Las tres son la misma función, pero PyTorch da pendientes de 0, 1 y 0,5 en cero.

Ejercicios

1. Una recta pasa por la esquina de la ReLU en cero con pendiente un medio. ¿Es su pendiente un subgradiente válido allí? ¿Y una recta con pendiente uno y medio?

Pista: Pruebe cada recta en una entrada de 2 y de menos 2: debe quedar por debajo de la ReLU, o tocarla, en todas partes.

Respuesta: Con pendiente un medio la recta vale 1 donde la ReLU vale 2, y menos 1 donde la ReLU vale 0, así que queda por debajo: es válida. Con pendiente uno y medio vale 3 donde la ReLU vale 2, por encima de la función, así que no es un subgradiente.

2. ¿Por qué hay que poner las estaciones en el orden adecuado antes del recorrido hacia atrás cuando un modelo se ramifica?

Pista: Piense en una estación cuya salida alimenta a dos estaciones posteriores.

Respuesta: Una estación solo puede pasar su gradiente cuando todo lo que depende de ella le ha devuelto su parte. Sin un orden adecuado, el recorrido podría pasar un gradiente incompleto y seguir un camino equivocado; micrograd lo evita con un paso de ordenación barato.

3. Describa con sus propias palabras qué hay que dar a una primitiva nueva para que un marco de trabajo pueda entrenar a través de ella.

Pista: Recuerde las dos puertas hacia atrás de cada primitiva.

Respuesta: Su pasada hacia delante, que calcula la salida, y sus VJP: uno que devuelve la fila viajera a la estación anterior y otro que da el gradiente de sus propios parámetros, si los tiene.