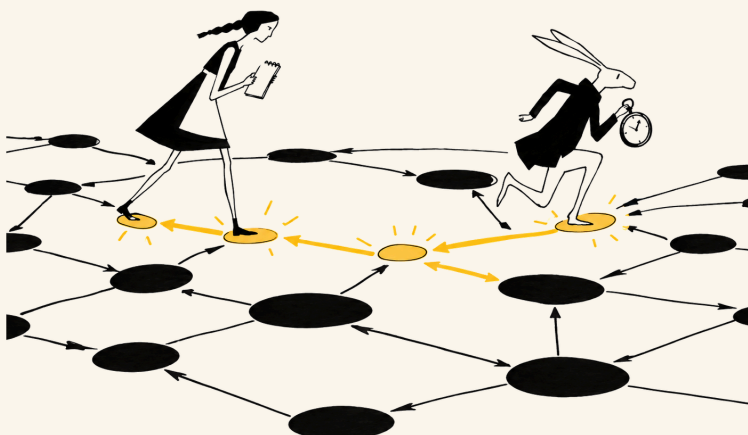


Las aventuras de Alicia en el País de las Maravillas Diferenciables



Simone Scardapane · 2024

Audiolecciones ilustradas

Lecturized

Las aventuras de Alicia en el País de las Maravillas Diferenciables

Audiolecciones ilustradas

Simone Scardapane · 2024

Contado como lecciones ilustradas por Lecturized

lecturized.com

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

Índice

Un programa que conoce sus propias pendientes

Un número al final, y el camino lento hacia delante

Recorrer la cadena hacia atrás

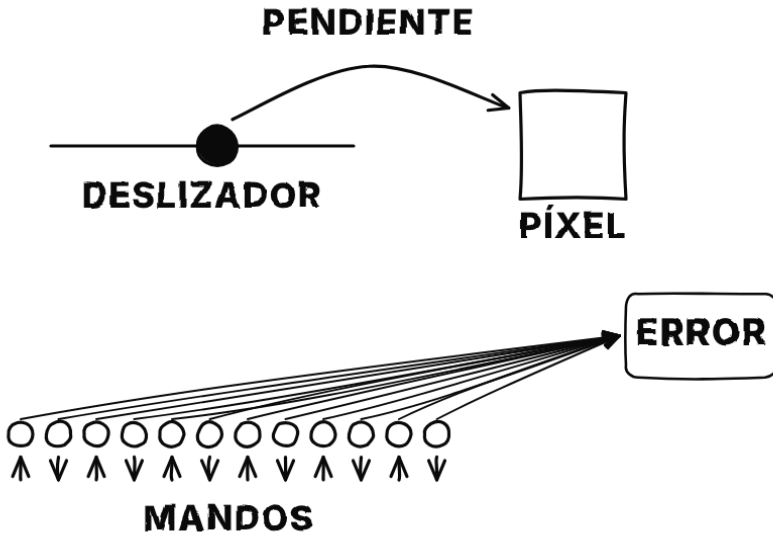
Cada pieza aprende a responder hacia atrás

Cómo lo hacen las bibliotecas, y por qué ganó la ReLU

La esquina de la rampa

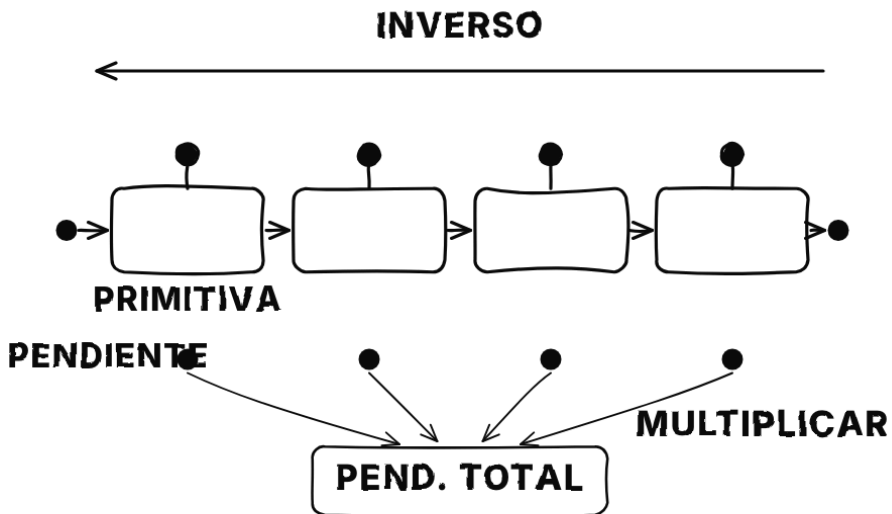
Un programa que conoce sus propias pendientes

Haga una foto con su teléfono y abra la aplicación de edición. Arrastre un poco a la derecha el deslizador del brillo y fíjese en un píxel del cielo: se aclara un poco, y usted nota cuánto lo cambia cada pequeño movimiento del deslizador. Esa sensación, «si giro un poco este mando, el resultado se mueve tanto», es lo que los matemáticos llaman una derivada. Ahora imagine no un deslizador sino cien millones, todos dentro del modelo que ordena sus fotos o responde a sus preguntas en el chat, y un único número al final que dice cuánto se equivocó el modelo con un lote de ejemplos. Entrenar significa girar un poco cada uno de esos mandos en la dirección correcta, una y otra vez, así que la pregunta que responde este capítulo es: ¿cómo averigua un ordenador, deprisa, hacia dónde girar cada mando?



Muchos mandos, un número

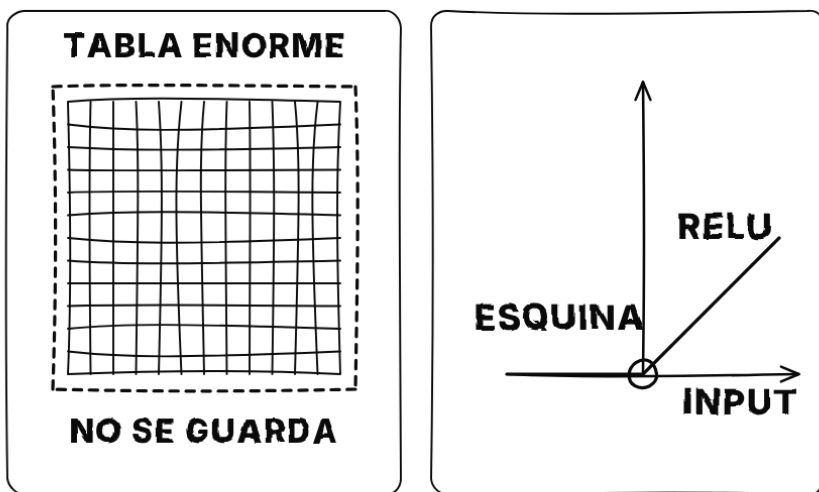
La respuesta del libro empieza por mirar el modelo como un programa. Cuando el programa se ejecuta, realiza una cadena de operaciones pequeñas y sencillas, cada una alimentando a la siguiente; el libro llama primitiva a cada una, y grafo computacional a toda la cadena tal como se ejecutó. Una primitiva puede ser tan pequeña como una multiplicación de una tabla de números, tan grande como una capa entera, o incluso un modelo entero, porque las piezas hechas de piezas siguen siendo piezas. Cada primitiva recibe algo, guarda algunos números ajustables propios, los pesos, y entrega algo.



Una cadena de primitivas

Todo descansa en una suposición modesta. Para cada primitiva ya sabemos cómo responde su salida a cambios pequeños, tanto en lo que entró como en sus propios pesos; el libro llama a estas dos tablas de pendientes el jacobiano de entrada y el jacobiano de los pesos. Conociendo las pendientes de cada pieza, la regla del cálculo escolar para una función dentro de otra, la regla de la cadena, nos da en principio la pendiente de todo el programa: basta multiplicar entre sí las pendientes de las piezas. Así que el problema parece resuelto antes de empezar, y toda la dificultad está en hacer esa multiplicación con la suficiente astucia para que sea rápida. La forma astuta se llama retropropagación, o diferenciación automática en modo inverso, y es el motor de todos los sistemas que entrenan a su asistente de chat.

El planteamiento trae dos advertencias. Las tablas de pendientes pueden ser enormes, porque cada número de entrada puede afectar a cada número de salida y se procesa a la vez todo un lote de fotos; un buen método debe usarlas sin escribirlas nunca completas. Y algunas piezas tienen esquinas, como la ReLU, que simplemente corta a cero los números negativos; el libro promete una forma de tratar esas esquinas más adelante en el capítulo.



Dos advertencias

Por último, el libro señala que la manera de escribir el programa determina cómo se encuentran las pendientes. En una biblioteca como JAX todo es una función, y los pesos se entregan como cualquier otra entrada, exactamente como en la notación del libro. En PyTorch o TensorFlow, una capa es un objeto que guarda sus pesos

dentro, lo que es cómodo, por ejemplo para crear los pesos solo cuando se conoce el tamaño de la entrada, pero deja los pesos a un paso de distancia de las matemáticas. Aquí se quedan fuera las formas de las tablas de pendientes para lotes de datos y cómo el libro las aplana en listas simples; las encontrará en el apartado sobre la notación.

Para recordar

Entrenar un modelo significa encontrar, para cada número ajustable, hacia dónde girarlo para que un único error final sea menor. El libro ve el modelo como un programa en ejecución, una cadena de primitivas sencillas, y supone conocidas las pendientes de cada pieza. La regla de la cadena da entonces la pendiente del conjunto, así que el verdadero reto es calcularla deprisa y sin guardar tablas gigantescas. Las bibliotecas de estilo funcional como JAX y las de estilo de objetos como PyTorch representan los pesos de forma distinta, y eso determina cómo se construye en cada una la diferenciación automática.

Preguntas

1. Si ya se conocen las pendientes de cada primitiva, ¿por qué hallar la pendiente de todo el programa sigue siendo un problema real?
 - a. Porque la regla de la cadena solo funciona para programas con una sola primitiva

- b. Porque multiplicar todas esas pendientes entre sí hay que hacerlo con astucia para que sea rápido
 - c. Porque las pendientes de las primitivas cambian cada vez que se ejecuta el programa
 - d. Porque el número del error final no puede calcularse con exactitud
2. ¿Por qué un buen método debe usar las tablas de pendientes sin escribirlas nunca completas?
- a. Porque las tablas pueden ser enormes: cada número de entrada puede afectar a cada número de salida, para todo un lote a la vez
 - b. Porque las tablas son partes secretas del modelo que la biblioteca oculta
 - c. Porque escribirlas cambiaría los pesos del modelo
 - d. Porque las tablas solo existen para las primitivas sin esquinas
3. ¿Qué necesita realmente de la diferenciación automática el entrenamiento del modelo que ordena sus fotos?
- a. El número del error final, calculado lo más deprisa posible
 - b. Una lista de todas las primitivas en el orden en que se ejecutaron
 - c. Para cada número ajustable, hacia dónde girarlo para que el error final sea menor
 - d. La forma exacta de cada tabla de pendientes
4. ¿En qué se diferencia PyTorch de JAX en la forma de representar los pesos de una capa?

- a. PyTorch entrega los pesos como cualquier otra entrada, JAX los guarda dentro de un objeto
- b. PyTorch no tiene pesos, solo entradas y salidas
- c. Los dos tratan los pesos exactamente como en la notación del libro
- d. PyTorch guarda los pesos dentro del objeto de la capa, JAX los entrega como cualquier otra entrada

Respuestas

- 1. **b)** La regla de la cadena dice que la pendiente del conjunto es el producto de las pendientes de las piezas, así que el problema parece resuelto; toda la dificultad está en hacer esa multiplicación con la suficiente astucia para que sea rápida.
- 2. **a)** Cada número de entrada puede afectar a cada número de salida, y se procesa a la vez todo un lote de fotos, así que las tablas de pendientes crecen demasiado para guardarlas.
- 3. **c)** Entrenar significa girar un poco cada uno de los muchos mandos en la dirección correcta, así que el ordenador debe averiguar, deprisa, hacia dónde girar cada uno.
- 4. **d)** En JAX todo es una función y los pesos se entregan como cualquier otra entrada, mientras que en PyTorch una capa es un objeto que guarda sus pesos dentro.

Ejercicios

- 1. Piense en la aplicación de edición de fotos con su deslizador de brillo. Explique con sus propias palabras qué le dice una derivada, y por qué un modelo con cien millones de deslizadores así necesita una forma rápida de hallarlas todas.

Pista: Empiece por lo que le pasa a un píxel cuando mueve un poco un deslizador.

Respuesta: Una derivada dice cuánto se mueve el resultado cuando se gira un poco un mando. El entrenamiento debe girar cada mando del modelo en la dirección correcta una y otra vez, así que necesita todas estas pendientes, para los cien millones de mandos, cada vez; hacerlo despacio haría imposible el entrenamiento.

2. Una primitiva puede ser una multiplicación, una capa entera o un modelo entero. Explique por qué el método de este capítulo sigue funcionando aunque las piezas sean de tamaños tan distintos.

Pista: Pregúntese qué necesita saber el método de cada pieza, sea cual sea su tamaño.

Respuesta: El método solo supone que, para cada pieza, sabemos cómo responde su salida a cambios pequeños en lo que entró y en sus propios pesos. Las piezas hechas de piezas siguen siendo piezas, así que una capa o un modelo entero pueden tratarse igual que una sola operación.

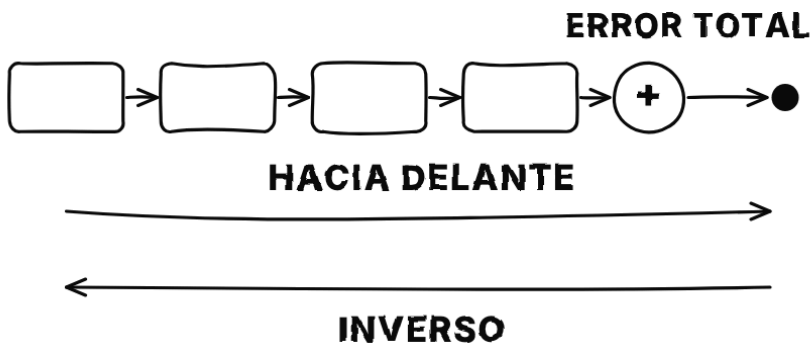
3. Algunas piezas, como la ReLU, tienen una esquina donde cortan a cero los números negativos. ¿Por qué podría preocupar una esquina así a un método basado en pendientes?

Pista: Imagine la gráfica de la ReLU justo en el punto donde se dobla.

Respuesta: En una esquina la gráfica no tiene una sola pendiente, así que la suposición de que se conoce la pendiente de cada pieza falla allí. El libro promete una forma de tratar esas esquinas más adelante en el capítulo.

Un número al final, y el camino lento hacia delante

Imagine la forma más sencilla que cualquiera usaría para averiguar hacia dónde girar un mando. Ejecuta el modelo con un lote de fotos y anota cuánto se equivocó; empuja un pelo un mando; vuelve a ejecutar todo el modelo y compara. La diferencia le da la pendiente para ese único mando, y funciona de verdad. Luego recuerda que el modelo de su aplicación de fotos tiene muchos millones de mandos, y cada uno pide su propio par de ejecuciones. Un solo paso de entrenamiento necesitaría millones de ejecuciones completas, y por eso nadie entrena un modelo así.



Empujar, repetir, un número

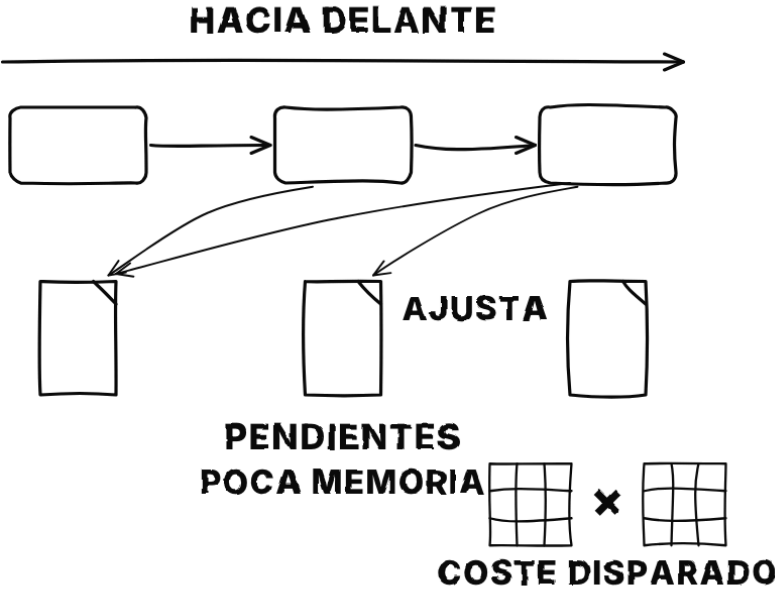
El libro plantea ahora la tarea con precisión. Un modelo en funcionamiento es una cadena de primitivas que se ejecutan una tras otra, cada una pasando su resultado a la siguiente; el último eslabón mide el error en cada ejemplo y un paso final suma esos errores para todo el lote. El libro llama traza de evaluación del programa a esta ejecución escrita entera. Lo que importa es que la cadena termina siempre en un único número, el error total, porque un número es lo que un optimizador puede hacer más pequeño. La diferenciación automática es entonces la tarea de hallar, todo a la vez y de forma eficiente, la pendiente de ese único número respecto de cada conjunto de pesos de la cadena, usando solo la propia cadena y las pendientes conocidas de cada pieza.

El libro añade que hay dos familias de métodos para esta tarea, el modo directo y el modo inverso, y que solo se diferencian en el orden en que se combinan las pendientes de las piezas. Ya aquí anuncia que el inverso, la retropropagación de las redes neuronales, es mucho mejor para nuestro propósito. Las mismas ideas se trasladan a cadenas que se ramifican y se vuelven a unir, y a modelos que usan los mismos pesos en varios sitios, algo que reaparece en el capítulo sobre modelos recurrentes.

Antes de tomar ninguno de los dos caminos, el libro aparta otras dos formas de derivar. La primera es el experimento que acaba de imaginar, empujar y volver a ejecutar, llamado diferenciación numérica; a dos ejecuciones por cada número, solo sirve como comprobación puntual de que un método más ingenioso da la respuesta correcta. La segunda es la diferenciación simbólica, la forma en que un programa de álgebra computacional hace cálculo de instituto: usted le da una fórmula y le devuelve la fórmula de la derivada. El libro muestra un ejemplo diminuto en unas pocas líneas de Python con la biblioteca SymPy: una fórmula corta construida con un seno, cuya derivada imprime el programa.

El problema del camino simbólico es el derroche. La fórmula de la derivada repite trozos de la original, el mismo seno y el mismo coseno, y un programa sensato calcularía cada uno una sola vez y lo guardaría. Encontrar la mejor forma posible de compartir ese trabajo, de modo que nada se calcule nunca dos veces, resulta ser uno de esos problemas para los que no se conoce ningún método general rápido. La diferenciación automática lo esquivo usando lo que ya tenemos, la cadena de pasos que el programa ejecutó de verdad, y así lo hace mucho mejor que el enfoque simbólico sin más.

Ahora, el primer camino: el modo directo. Imagínelo como recorrer la cadena en el sentido en que funciona el programa, llevando un cuaderno. Cada vez que se ejecuta una pieza nueva, abre una página nueva para sus pesos, anotando cómo depende de ellos su salida, y además vuelve atrás y actualiza todas las páginas que ha escrito hasta ahora, porque la pieza nueva transmite todas las influencias anteriores. Al llegar al final, sumar los errores finales convierte cada página en la pendiente que buscaba. Lo bueno es que esta contabilidad avanza junto al programa, así que necesita más o menos tanta memoria como el propio programa.



Modo directo: un cuaderno

Lo malo es el precio de cada actualización. Actualizar una página significa multiplicar entre sí dos tablas enteras de pendientes, y el

tamaño de ese trabajo crece con el cuadrado del tamaño del lote y con el cuadrado del número de características de cada ejemplo. Para un lote de fotos, donde cada ejemplo tiene muchos miles de valores de píxeles, el coste se dispara, y vuelve a dispararse por cada página anterior que hay que actualizar. Igual que el experimento de empujar mandos, el modo directo es correcto, y demasiado lento para los modelos de su teléfono. Aquí se quedan fuera las fórmulas exactas de las actualizaciones y la estimación precisa del coste; están en la sección sobre la diferenciación en modo directo.

Para recordar

La ejecución de un modelo es una cadena de pasos que termina siempre en un número, el error total en el lote, y la diferenciación automática debe hallar la pendiente de ese número para cada conjunto de pesos a la vez. Empujar cada mando y volver a ejecutar funciona, pero cuesta dos ejecuciones por número, así que solo sirve como comprobación, y la diferenciación simbólica sin más malgasta esfuerzo en trozos repetidos. El modo directo recorre la cadena en el orden de ejecución, actualizando en cada paso una página de pendientes para cada conjunto de pesos anterior. Necesita poca memoria, pero cada actualización multiplica tablas enteras, un coste que se dispara con lotes grandes e imágenes, así que hace falta un orden mejor.

Preguntas

1. ¿Por qué la cadena de pasos del entrenamiento termina siempre en un único número?
 - a. Porque un ordenador solo puede guardar un número al final de un programa
 - b. Porque un número, el error total, es lo que un optimizador puede hacer más pequeño
 - c. Porque cada primitiva de la cadena da un solo número
 - d. Porque el último paso del modelo es siempre una verificación cruzada
2. ¿Por qué empujar cada mando y volver a ejecutar el modelo solo sirve como comprobación puntual?
 - a. Porque da pendientes erróneas para la mayoría de los mandos
 - b. Porque solo funciona para modelos sin pesos
 - c. Porque necesita dos ejecuciones por cada número, millones de ejecuciones por paso de entrenamiento
 - d. Porque no puede usarse con un lote de fotos
3. ¿Qué tiene de derrochadora la diferenciación simbólica sin más?
 - a. La fórmula de la derivada repite trozos, como el mismo seno y el mismo coseno, que deberían calcularse una sola vez
 - b. Solo da una derivada aproximada
 - c. Necesita una nueva ejecución del modelo por cada peso
 - d. No puede tratar una fórmula que contenga un seno
4. El modo directo necesita poca memoria. Entonces, ¿por qué es demasiado lento para un modelo de fotos?

- a. Porque tiene que ejecutar el modelo entero una vez por mando
- b. Porque cada actualización multiplica tablas enteras, con un coste que crece con el cuadrado del tamaño del lote y de las características
- c. Porque recorre la cadena hacia atrás y debe guardar cada paso
- d. Porque solo puede tratar un ejemplo cada vez

Respuestas

1. **b)** El último eslabón mide el error en cada ejemplo y un paso final los suma; un número es lo que un optimizador puede hacer más pequeño.
2. **c)** La diferenciación numérica cuesta dos ejecuciones por número, así que con millones de mandos un paso de entrenamiento necesitaría millones de ejecuciones completas.
3. **a)** La fórmula de la derivada repite trozos de la original, y encontrar la mejor forma de compartir ese trabajo es un problema sin ningún método general rápido conocido.
4. **b)** Actualizar una página significa multiplicar dos tablas enteras de pendientes, y ese trabajo crece con el cuadrado del tamaño del lote y del número de características, y se repite para cada página anterior.

Ejemplos resueltos

Contar ejecuciones al empujar mandos La diferenciación numérica necesita dos ejecuciones del modelo por cada número cuya pendiente queremos. ¿Cuántas ejecuciones completas necesitaría un paso de entrenamiento para un modelo con 1 000 000 de pesos?

Cada peso es un número cuya pendiente necesitamos, así que hay 1 000 000 de pendientes que hallar.

Cada pendiente cuesta dos ejecuciones del modelo entero.

Así que el total es $2 \times 1\,000\,000$ ejecuciones.

Respuesta: 2 000 000 de ejecuciones completas para un solo paso de entrenamiento

Ejercicios

1. En el modo directo, el coste de cada actualización crece con el cuadrado del tamaño del lote. Si duplica el lote de fotos de 32 a 64, ¿por qué factor crece el coste de una actualización?

Pista: Elevar al cuadrado: ¿qué le pasa al cuadrado de un número cuando el número se duplica?

Respuesta: Por un factor de 4: el coste va con el cuadrado del tamaño del lote, y duplicar un número hace su cuadrado cuatro veces mayor ($64^2 / 32^2 = 4$).

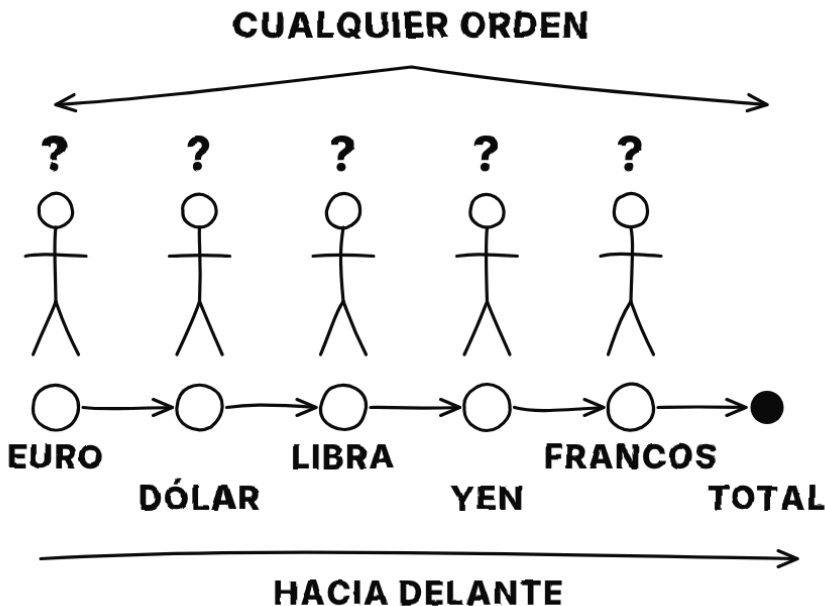
2. Explique con sus propias palabras la imagen del cuaderno del modo directo: ¿qué se escribe en una página nueva, y por qué hay que actualizar las páginas anteriores en cada paso?

Pista: Piense en lo que hace cada paso nuevo con la influencia de todos los pesos anteriores sobre el resultado.

Respuesta: Cuando se ejecuta una pieza nueva, una página nueva anota cómo depende su salida de sus propios pesos. También hay que actualizar todas las páginas anteriores, porque la pieza nueva transmite la influencia de todos los pesos anteriores, de modo que su efecto sobre el resultado cambia en cada paso.

Recorrer la cadena hacia atrás

Ponga a cinco amigos en fila y juegue a los tipos de cambio. El primero convierte sus euros en dólares, el siguiente los dólares en libras, luego en yenes, luego en francos, y el último le dice en cuántos francos se convierte al final un euro. Puede multiplicar los cinco tipos de cambio en cualquier orden y obtener la misma respuesta, porque a la multiplicación no le importa el orden. Pero suponga que quiere saber, para cada amigo de la fila, cuánto movería el número final un cambio en su tipo de cambio; entonces el orden del trabajo de repente importa muchísimo.

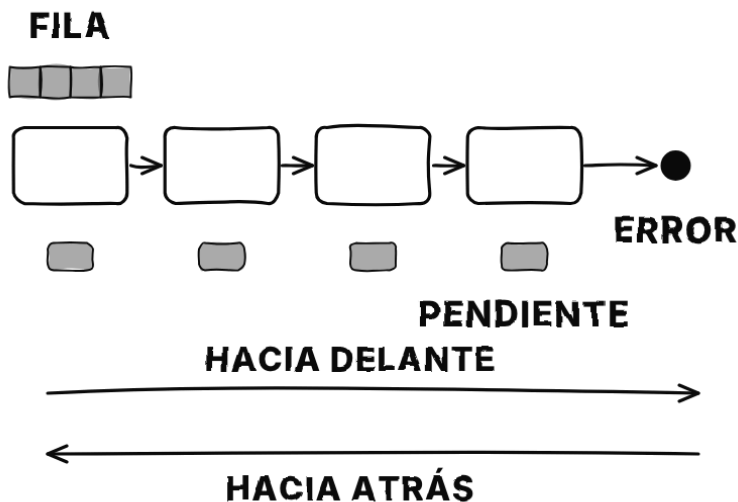


Cinco amigos, cinco tipos de cambio

Este es el giro que da el libro en su sección sobre el modo inverso.

Escrita entera, la pendiente del error final respecto de un conjunto de pesos es una larga fila de tablas de pendientes multiplicadas entre sí: las tablas de todos los pasos posteriores y, en el extremo, la tabla de pesos de ese paso. Como la multiplicación puede agruparse en cualquier orden, somos libres de elegir. El modo directo, como vimos, trabajaba en el orden en que se ejecutaba el programa, y lo pagaba multiplicando tablas enteras una y otra vez.

La observación clave del libro es que la cadena termina en un número, el error total. Así que, si empieza la multiplicación por ese extremo, lo que lleva nunca es una tabla entera, solo una fila de números, y una fila por una tabla es barato y da otra fila. Mejor aún: la fila de un paso es simplemente la fila del paso siguiente, multiplicada una vez más; así que puede construirlas todas recorriendo la cadena hacia atrás, paso a paso. Eso es la diferenciación automática en modo inverso, la retropropagación. Primero todo el programa se ejecuta hacia delante y guarda cada resultado intermedio; luego un programa en espejo vuelve desde el error hasta el primer paso, y en cada paso entrega la pendiente para los pesos de ese paso y pasa la fila más atrás. A las filas que lleva, el libro las llama adjuntos: cada uno dice con qué fuerza responde el error final a un resultado intermedio.

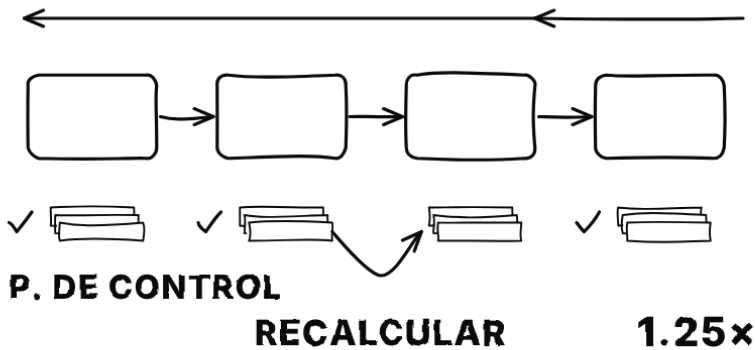


Llevar una fila hacia atrás

En el lenguaje de las redes neuronales, la ejecución normal es la pasada hacia delante, que no hay que confundir con el modo directo, y el camino de vuelta es la pasada hacia atrás. Cada operación de la pasada hacia atrás es una fila por una tabla, cuyo coste crece solo en proporción a los tamaños en juego, así que el modo inverso es mucho más rápido que el directo. Cuando se entrenó el asistente de su teléfono, cada uno de sus incontables pasos de entrenamiento fue una pasada hacia delante y una pasada hacia atrás como esta.

El precio es la memoria. La pasada hacia atrás necesita cada resultado intermedio de la pasada hacia delante, así que hay que guardarlos todos hasta que el camino de vuelta llegue a ellos. Los puntos de control del gradiente alivian esto guardando solo unos pocos

resultados, los puntos de control, y recalculando el resto cuando hace falta. La figura del libro muestra una cadena de cuatro bloques: solo se guardan las salidas primera, segunda y cuarta; la pasada hacia atrás se detiene en la tercera, vuelve a ejecutar hacia delante desde el punto de control más cercano para reconstruir lo que faltaba y luego sigue, con una cuarta parte más de cálculo en este ejemplo. Cuantos menos puntos de control guarde, más recalculará: se cambia memoria por tiempo. Aquí se quedan fuera las fórmulas del programa inverso y el algoritmo paso a paso; están en la sección sobre la diferenciación en modo inverso.



Puntos de control: memoria por tiempo

Para recordar

La pendiente del error para cualesquiera pesos es un largo producto de tablas, y el orden de ese producto lo elegimos nosotros. Como la cadena termina en un número, empezar por el final significa llevar solo una fila, lo que es barato, y construir cada fila a partir de la siguiente recorriendo la cadena hacia atrás. Eso es la retropropagación: una pasada hacia delante que guarda cada resultado intermedio, y luego una pasada hacia atrás que entrega todas las pendientes. Es mucho más rápida que el modo directo pero necesita mucha memoria, que los puntos de control reducen recalculando algunos resultados en lugar de guardarlos.

Preguntas

1. ¿Por qué empezar la multiplicación por el final de la cadena hace barato el modo inverso?
 - a. Porque las últimas tablas de la cadena son siempre las más pequeñas
 - b. Porque la cadena termina en un número, así que lo que se lleva es solo una fila, y una fila por una tabla es barato
 - c. Porque la pasada hacia atrás se salta los pasos que no tienen pesos
 - d. Porque multiplicar desde el final da una respuesta más precisa

2. ¿Qué tiene que ocurrir antes de que pueda empezar la pasada hacia atrás?
 - a. Todo el programa debe ejecutarse hacia delante y guardar cada resultado intermedio
 - b. Cada peso debe empujarse una vez para comprobar su pendiente
 - c. Deben multiplicarse entre sí las tablas de pendientes de todos los pasos
 - d. El usuario debe elegir los puntos de control para cada paso
3. ¿Qué intercambian los puntos de control del gradiente?
 - a. Precisión de las pendientes por velocidad de la pasada hacia delante
 - b. Velocidad de la pasada hacia atrás por un error final más pequeño
 - c. Memoria por cálculo extra: menos resultados guardados, más recalculados
 - d. La pasada hacia delante por una segunda pasada hacia atrás
4. En el juego de los tipos de cambio, ¿por qué importa el orden del trabajo si a la multiplicación no le importa el orden?
 - a. Porque los tipos de cambio varían mientras se multiplica
 - b. Porque solo el tipo de cambio del último amigo afecta al número final
 - c. Porque el número final es distinto según el orden
 - d. Porque se quiere el efecto del tipo de cambio de cada amigo, y algunos órdenes permiten reutilizar el trabajo mucho mejor que otros

Respuestas

1. **b)** La cadena termina en un número, el error total, así que empezando por ese extremo solo se lleva una fila, y una fila por una tabla es barato y da otra fila.
2. **a)** Primero todo el programa se ejecuta hacia delante y guarda cada resultado intermedio; solo entonces el programa en espejo vuelve desde el error.
3. **c)** Los puntos de control guardan solo unos pocos resultados y recalculan el resto cuando hace falta; cuantos menos puntos de control, más se recalcula, así que se cambia memoria por tiempo.
4. **d)** La respuesta es la misma en cualquier orden, pero cuando se necesita la influencia de cada paso, elegir el orden decide cuánto trabajo se repite.

Ejemplos resueltos

El coste de un hueco entre puntos de control Una cadena tiene cuatro bloques. Con puntos de control, solo se guardan las salidas del primer, segundo y cuarto bloque. Durante la pasada hacia atrás falta la salida del tercer bloque, que hay que reconstruir volviendo a ejecutar hacia delante desde el punto de control más cercano. ¿Cuánto cálculo más hace falta en comparación con guardarlo todo?

La pasada hacia delante normal ejecuta los cuatro bloques una vez.

La pasada hacia atrás se detiene en el tercer bloque, cuyo resultado no se guardó.

Volver a ejecutar hacia delante desde el punto de control más cercano, la salida del segundo bloque, significa ejecutar el tercer bloque una vez más.

Así que hay el trabajo de un bloque extra sobre cuatro, es decir, el de cinco bloques en lugar de cuatro.

Respuesta: 1,25 veces el cálculo, una cuarta parte más, a cambio de no guardar el resultado de un bloque

Ejercicios

1. Si no guardara ningún punto de control salvo la entrada, ¿qué pasaría con la memoria y con el cálculo durante la pasada hacia atrás?

Pista: Pregúntese qué hay que reconstruir cada vez que la pasada hacia atrás necesita un resultado que falta.

Respuesta: La memoria sería muy pequeña, pero cada resultado que faltara habría que reconstruirlo ejecutando hacia delante desde el principio, así que el cálculo crecería mucho. Cuantos menos puntos de control se guardan, mayor es el coste de la pasada hacia atrás.

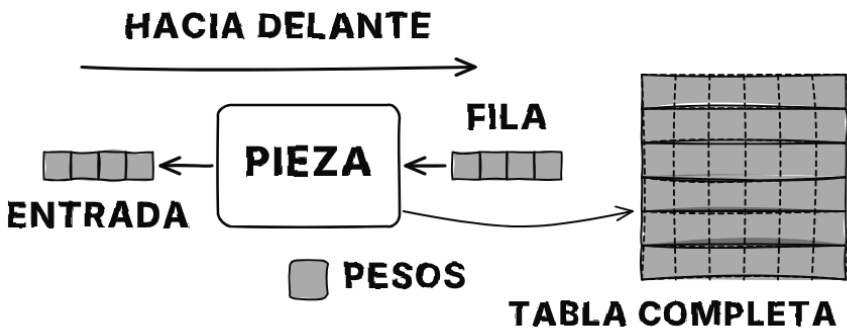
2. Explique con sus propias palabras qué es un adjunto y por qué cada uno puede construirse a partir del siguiente a lo largo de la cadena.

Pista: Piense en qué mide la fila que se lleva hacia atrás, y qué le añade un paso más hacia atrás.

Respuesta: Un adjunto dice con qué fuerza responde el error final a un resultado intermedio. Ir un paso más atrás solo significa multiplicar una vez más por la tabla de pendientes de ese paso, así que cada adjunto sale del que va después, y al recorrer la cadena hacia atrás se construyen todos.

Cada pieza aprende a responder hacia atrás

Pregunte a un dependiente cómo cambiaría el precio de un pastel si la harina costara un céntimo más. No necesita enseñarle toda la hoja de costes de la pastelería; uno bueno simplemente responde «medio céntimo, más o menos», porque sabe lo que lleva el pastel. Ese es el espíritu de esta sección: durante el camino de vuelta, nadie necesita ver las tablas completas de pendientes, solo la respuesta que da cada pieza cuando se le entrega la fila que vuelve.



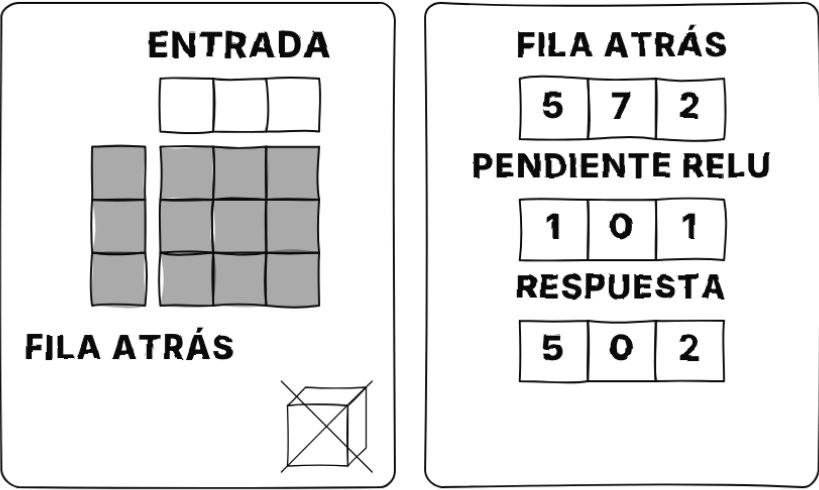
Dos respuestas de vuelta

Mire otra vez lo que hace realmente la pasada hacia atrás en cada paso. Toma la fila que lleva, el adjunto, y la multiplica por una tabla de la pieza actual: una vez por la tabla de los pesos de la pieza, para obtener su pendiente, y otra por la tabla de la entrada de la pieza, para obtener la fila del paso anterior. Una fila por una tabla: eso es todo. El libro da un nombre a esta operación, el producto vector-jacobiano, y cada pieza tiene dos, uno para su entrada y otro para sus pesos.

De aquí sale el principal resultado práctico. Para construir todo un sistema de diferenciación automática, se elige un conjunto de primitivas y se enseña a cada una, además de su trabajo normal hacia delante, a responder estas dos preguntas hacia atrás. Las tablas completas de pendientes no tienen que escribirse en memoria en ningún momento. La figura del libro muestra una primitiva como una caja con su flecha hacia delante en un sentido y dos pequeñas respuestas hacia atrás que salen en el otro, una para la entrada y otra para los pesos. Y no se pierde nada: si alguna vez quiere la tabla completa, puede reconstruirla fila a fila haciendo la pregunta hacia atrás con filas de prueba sencillas.

Dos ejemplos muestran por qué las respuestas son tanto más baratas que las tablas. Para la pieza más común de una red, multiplicar la entrada por una tabla de pesos, la tabla completa de pendientes respecto de los pesos sería un enorme bloque de números de tres dimensiones. Y sin embargo, las respuestas hacia atrás son sencillas: la fila que se devuelve es simplemente la fila que llega pasada por los mismos pesos puestos de lado, y la pendiente de los pesos es una cuadrícula simple hecha con dos listas, la fila que llega y la entrada, en la que cada casilla es el producto de una pareja. Para las piezas tipo interruptor entre capas, las funciones de activación como la ReLU, que actúan sobre cada número por separado, la tabla completa sería

casi entera de ceros; la respuesta hacia atrás consiste simplemente en escalar cada número de la fila que llega por la pendiente de esa pieza en ese punto. Cada vez que se entrenó el modelo de fotos de su teléfono, su pasada hacia atrás fue una larga secuencia de exactamente estas pequeñas respuestas. Aquí se quedan fuera las fórmulas en sí y la breve deducción de la respuesta de los pesos; están en el apartado sobre productos vector-jacobiano.



Respuestas baratas, dos ejemplos

Para recordar

Cada paso de la pasada hacia atrás es el mismo movimiento: la fila que vuelve por una tabla de la pieza actual. Estos movimientos se llaman productos vector-jacobiano, y cada pieza necesita dos, uno para su entrada y otro para sus pesos. Así que un sistema de diferenciación automática es un conjunto de piezas que saben responder cada una a estas dos preguntas hacia atrás, y las tablas completas de pendientes nunca se guardan. Para una multiplicación por pesos, las respuestas son los pesos puestos de lado y una cuadrícula de productos; para una activación, un simple escalado de cada número.

Preguntas

1. ¿Por qué un sistema de diferenciación automática puede funcionar sin escribir nunca las tablas completas de pendientes?
 - a. Porque las tablas son todo ceros salvo en la primera fila
 - b. Porque cada pieza solo tiene que responder cuánto vale la fila que llega por su tabla, y esa respuesta puede calcularse directamente
 - c. Porque la pasada hacia atrás solo necesita la pendiente de la última pieza
 - d. Porque las tablas se calculan una vez y se tiran antes del entrenamiento

2. ¿Cuántas respuestas hacia atrás tiene que dar una pieza con pesos, y para qué sirven?
- a. Una, la pendiente del error final
 - b. Tres, una para la entrada, otra para los pesos y otra para la salida
 - c. Dos: una para su entrada, que da la fila del paso anterior, y otra para sus pesos, que da su pendiente
 - d. Ninguna, el entorno las calcula solo a partir de la pasada hacia delante
3. ¿Cuál es la respuesta hacia atrás de una función de activación como la ReLU?
- a. Escalar cada número de la fila que llega por la pendiente propia de la pieza en ese punto
 - b. Multiplicar la fila que llega por una tabla enorme de pesos
 - c. Poner de lado la fila que llega y pasarla sin cambios
 - d. Hacer una cuadrícula de productos con la fila que llega y la entrada

Respuestas

- 1. b)** Cada paso de la pasada hacia atrás es la fila que vuelve por una tabla, y cada pieza puede responder a eso directamente, así que las tablas completas nunca se guardan.
- 2. c)** La pasada hacia atrás multiplica la fila una vez por la tabla de los pesos, para la pendiente de los pesos, y otra por la tabla de la entrada, para la fila que se pasa más atrás.
- 3. a)** Una activación actúa sobre cada número por separado, así que su tabla completa es casi todo ceros y la respuesta es un simple escalado de cada número.

Ejemplos resueltos

La cuadrícula de productos de los pesos Una pieza multiplica su entrada por una tabla de pesos. La entrada es la lista $(1, 4)$ y la fila que vuelve es $(2, 3)$. El libro muestra que la pendiente del peso de la fila i y la columna j es el i -ésimo número de la fila que llega por el j -ésimo número de la entrada. Construya la cuadrícula de pendientes de los pesos.

Cada casilla es el producto de una pareja: el número de la fila que llega para su fila, el número de la entrada para su columna.

Primera fila, con 2 llegando: $2 \times 1 = 2$ y $2 \times 4 = 8$.

Segunda fila, con 3 llegando: $3 \times 1 = 3$ y $3 \times 4 = 12$.

No hizo falta ningún bloque de tres dimensiones, solo estos cuatro productos.

Respuesta: La cuadrícula es 2, 8 en la primera fila y 3, 12 en la segunda

Ejercicios

1. Una pieza ReLU recibió las entradas $(-1, 2, 5)$. Su pendiente es 0 donde la entrada era negativa y 1 donde era positiva. La fila que vuelve es $(4, 7, 2)$. ¿Qué fila pasa la ReLU más atrás?

Pista: Escale cada número de la fila que llega por la pendiente de la ReLU en la entrada correspondiente.

Respuesta: $(0, 7, 2)$: la primera entrada era negativa, así que su pendiente es 0 y el 4 se convierte en 0; las otras dos pendientes son 1, así que el 7 y el 2 pasan sin cambios.

2. Si alguien aún quisiera la tabla completa de pendientes de una pieza, ¿cómo podría obtenerla usando solo su respuesta hacia atrás?

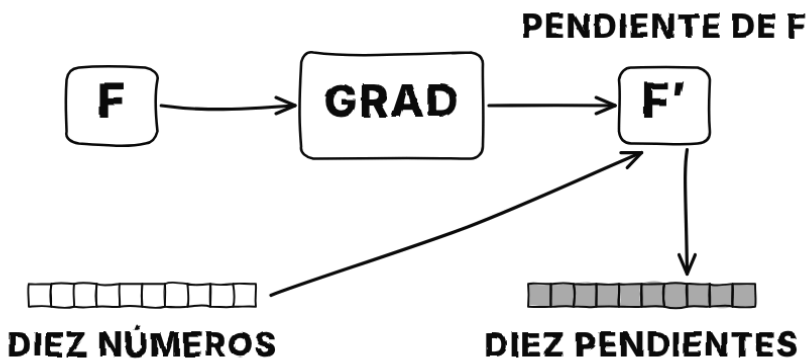
Pista: Piense en hacer la pregunta hacia atrás con filas de prueba muy sencillas.

Respuesta: Hacer la pregunta hacia atrás una vez por cada fila de prueba sencilla que tenga un solo 1 y ceros en lo demás; cada respuesta es una fila de la tabla completa, así que la tabla se reconstruye fila a fila.

Cómo lo hacen las bibliotecas, y por qué ganó la ReLU

Abra la calculadora de su teléfono y escriba un cuarto, 0,25.

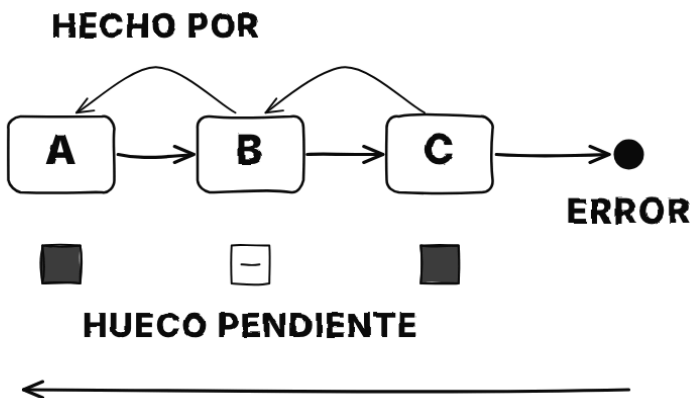
Multiplíquelo por 0,25, y otra vez, y otra, diez veces en total. Quizá espere un número pequeño; lo que obtiene es más o menos uno entre un millón, y tras veinte pasos así, uno entre un billón. Retenga ese desplome, porque decide qué funciones de activación puede usar una red profunda, y cierra esta parte.



Un cuarto, y luego fábricas de funciones

Primero, el libro mira cómo organizan la pasada hacia atrás las bibliotecas reales. Hay varias formas de construirla, desde llevar una lista en curso de cada operación, como hace TensorFlow, hasta reescribir el propio código del programa en una versión hacia atrás. En una biblioteca de estilo funcional como JAX, la idea de la parte anterior se vuelve muy literal: la respuesta hacia atrás es una función que recibe una función y un punto, y devuelve una función nueva que espera la fila que llega. Del mismo modo, el propio gradiente es una función que recibe su función y devuelve otra función, que calcula la pendiente en cualquier punto que le dé. El pequeño ejemplo del libro hace exactamente esto para la suma de los cuadrados de diez números: usted pide su gradiente y recibe una función cuya respuesta, en cualesquiera diez números, es otra vez una lista de diez pendientes. PyTorch ofrece el mismo estilo en un módulo propio, y combinando estas fábricas de funciones se pueden calcular cosas más avanzadas, como las pendientes de las pendientes.

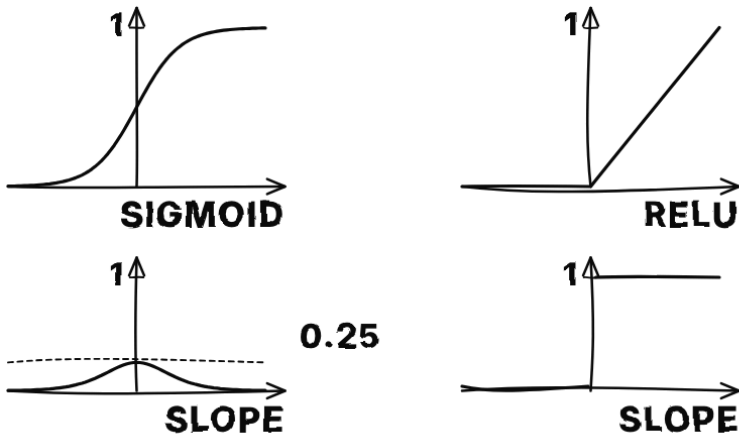
Los modelos de estilo de objetos, la forma habitual de escribir en PyTorch, necesitan un paso más. Una opción es sacar los pesos del objeto del modelo y llamarlo como una función simple a la que se entregan esos pesos. Lo más habitual es que PyTorch haga el trabajo entre bastidores: cada resultado que calcula recuerda qué operación lo creó, y tiene un hueco para su propia pendiente, vacío al principio. Cuando usted pide la pasada hacia atrás sobre el error final, PyTorch sigue esos recuerdos hacia atrás por la cadena y llena los huecos de cada magnitud que pidió guardar una pendiente; solo de esas, para no malgastar memoria. Cada vez que ve un modelo ajustándose en un portátil, esa llamada a la pasada hacia atrás está haciendo exactamente este recorrido.



Resultados que recuerdan

Volvamos ahora a la calculadora. En la pasada hacia atrás, cada función de activación multiplica la fila que vuelve por su propia pendiente. Si esa pendiente está por debajo de uno en todas partes, una red profunda multiplica la fila por un número pequeño una y otra vez, y las pendientes de las primeras capas se encogen hacia cero: el problema del desvanecimiento del gradiente. Si está por encima de uno en todas partes, crecen sin límite: el problema de la explosión del gradiente. No son solo cuestiones teóricas, porque los ordenadores guardan los números con una precisión limitada, así que los números que se vuelven demasiado pequeños o demasiado grandes simplemente se pierden tras no muchas capas. Una curiosidad que cuenta el libro: precisamente por esa precisión limitada, una pila de

capas puramente lineales no es perfectamente lineal, y esto incluso se ha usado para entrenar redes así.



La sigmoide encoge, la ReLU deja pasar

La sigmoide, una suave curva en forma de S que fue popular en el pasado, muestra el problema en su peor versión: su pendiente nunca pasa de un cuarto, el número que acaba de multiplicar por sí mismo. La figura del libro dibuja la curva en S junto con su pendiente, una joroba baja que llega a un cuarto en el centro y se aplana hasta nada a ambos lados. Una activación perfecta tendría pendiente exactamente uno en todas partes, pero esa es la función que no hace nada, y no separa una capa de la siguiente. La ReLU es el término medio: para entradas negativas su pendiente es cero, así que allí la fila simplemente se apaga, y para entradas positivas su pendiente es uno,

así que la fila pasa sin cambios, sin encogerse ni crecer. Su figura es una línea plana que se convierte en una rampa recta, y su pendiente, un escalón de cero a uno. Un regalo más: como la ReLU solo borra los negativos, su pendiente puede leerse en su salida igual que en su entrada, así que no hace falta guardar la entrada, y las bibliotecas pueden sobrescribirla en el sitio para ahorrar memoria. Aquí se quedan fuera las fórmulas exactas hacia atrás y el código de estas bibliotecas; los da el apartado sobre la implementación del modo inverso.

Para recordar

Las bibliotecas construyen la pasada hacia atrás de formas distintas: JAX convierte los gradientes en funciones que devuelven funciones, y PyTorch hace que cada resultado recuerde la operación que lo produjo y llene su propio hueco de pendiente durante la pasada hacia atrás. En la pasada hacia atrás, cada activación multiplica la fila por su pendiente, así que una pendiente menor que uno en todas partes hace desvanecerse los gradientes en las redes profundas, y una mayor que uno los hace explotar. La pendiente de la sigmoide es como mucho un cuarto, y por eso falla en las redes profundas. La ReLU deja pasar la fila sin cambios o no la deja pasar en absoluto, lo que mantiene sanos los gradientes, y hasta permite a las bibliotecas ahorrar memoria sobrescribiendo su entrada.

Preguntas

1. ¿Por qué las redes profundas construidas con la sigmoide sufren el desvanecimiento del gradiente?
 - a. Porque la pendiente de la sigmoide es como mucho un cuarto, así que la fila que vuelve se encoge en cada capa
 - b. Porque la sigmoide no tiene ninguna pendiente para entradas negativas
 - c. Porque la pendiente de la sigmoide es mayor que uno, así que los números desbordan
 - d. Porque la sigmoide no puede calcularse con precisión limitada
2. ¿Por qué no usar una activación cuya pendiente sea exactamente uno en todas partes?
 - a. Porque los ordenadores no pueden guardar el número uno con exactitud
 - b. Porque una función así haría explotar los gradientes
 - c. Porque la única función así no hace nada y no separa una capa de la siguiente
 - d. Porque su respuesta hacia atrás necesitaría una tabla completa de pendientes
3. ¿Cómo sabe PyTorch adónde ir durante la pasada hacia atrás de un modelo de estilo de objetos?
 - a. Vuelve a ejecutar el modelo en el orden inverso de las líneas de código
 - b. Cada resultado recuerda la operación que lo creó, y la pasada hacia atrás sigue esos recuerdos
 - c. Le pide al usuario que enumere las operaciones del modelo

- d. Convierte el modelo en una lista de pesos y deriva cada uno empujándolo
4. ¿Por qué una biblioteca puede sobrescribir la entrada de una ReLU sin estropear la pasada hacia atrás?
- a. Porque la ReLU no tiene respuesta hacia atrás
 - b. Porque la pendiente de la ReLU es siempre uno
 - c. Porque la pasada hacia atrás nunca pasa por las funciones de activación
 - d. Porque la ReLU solo borra los negativos, así que su pendiente también puede leerse en su salida

Respuestas

1. **a)** Cada activación multiplica la fila que vuelve por su pendiente, y la pendiente de la sigmoide nunca pasa de un cuarto, así que muchas capas encogen hacia cero las pendientes de las primeras capas.
2. **c)** Una pendiente de uno en todas partes significa la función que no hace nada en absoluto, que no separa las capas; la ReLU es el término medio.
3. **b)** Cada resultado que calcula PyTorch recuerda qué operación lo creó, y la pasada hacia atrás sigue esos recuerdos por la cadena, llenando los huecos de pendiente.
4. **d)** La ReLU deja sin cambios los valores positivos y borra los negativos, así que su pendiente es la misma leída en la entrada o en la salida, y no hace falta guardar la entrada.

Ejemplos resueltos

Diez capas sigmoides en el mejor caso La pendiente de la sigmoide nunca pasa de un cuarto. En la pasada hacia atrás, cada capa sigmoide multiplica la fila que vuelve por su pendiente. Incluso en el

mejor caso, ¿por qué factor se encoge una pendiente después de volver a través de diez capas sigmoides?

El mejor caso es la pendiente más grande, un cuarto, en cada capa.

Cada capa multiplica por 0,25, así que diez capas multiplican por 0,25 diez veces.

0,25 elevado a diez es unos 0,00000095, más o menos uno entre un millón.

Respuesta: Al menos un millón de veces, más o menos, incluso en el mejor caso

Ejercicios

1. Una fila que vuelve pasa por cinco capas ReLU, y en cada capa la entrada del número que seguimos era positiva. ¿Por qué factor se ha multiplicado ese número?

Pista: ¿Cuál es la pendiente de la ReLU para una entrada positiva?

Respuesta: Por exactamente uno: la pendiente de la ReLU para entradas positivas es uno, y uno multiplicado por sí mismo cinco veces sigue siendo uno, así que el número vuelve sin cambios, sin encogerse ni crecer.

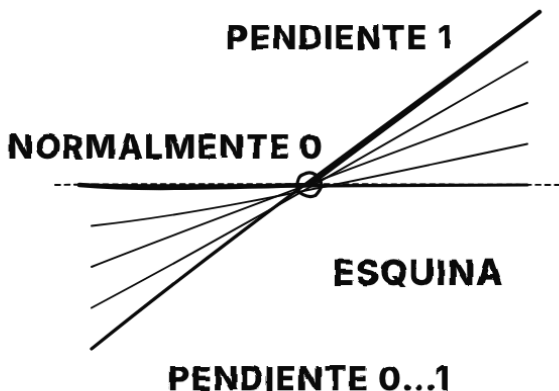
2. El ejemplo del libro pide el gradiente de la suma de los cuadrados de diez números. Cuando llama a la función gradiente en unos diez números, ¿cuántos números devuelve, y por qué?

Pista: El gradiente contiene una pendiente por cada número que recibe la función.

Respuesta: Diez: el gradiente da una pendiente por cada uno de los diez números de entrada, así que la respuesta vuelve a ser una lista de diez.

La esquina de la rampa

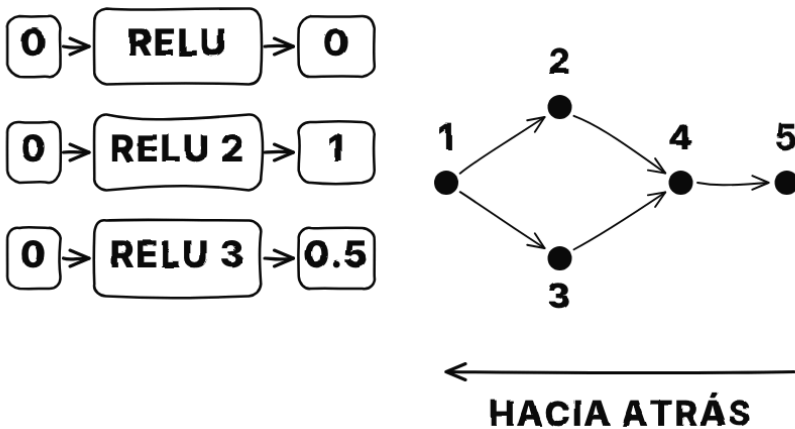
Ponga una regla en el suelo junto al pie de una rampa, donde el suelo plano se encuentra con la pendiente. Puede inclinar la regla para que quede plana sobre el suelo, o levantarla a lo largo de la rampa, o en cualquier posición intermedia, y en todas sigue tocando la esquina sin dejar de estar por debajo del suelo y de la rampa. Sobre el suelo liso, o sobre la rampa lisa, la regla solo puede apoyarse de una manera; en la esquina tiene todo un abanico de maneras. Ese es exactamente el problema de la ReLU, cuya gráfica es un suelo plano y después una rampa recta, y es la última pregunta del capítulo: ¿qué pendiente usamos en la esquina?



Un abanico de reglas en la esquina

Primero viene la respuesta práctica. El entrenamiento empieza con números aleatorios y avanza en pasos pequeños y ruidosos, así que la probabilidad de caer exactamente en la esquina, en una entrada exactamente igual a cero, es prácticamente nula; en todos los demás puntos la pendiente está perfectamente definida. La respuesta técnica es el subgradiente: cualquier pendiente de una recta que toca la curva en un punto y queda por debajo de toda la curva, como la regla. Donde una curva es suave solo hay una recta así, la derivada 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 las bibliotecas simplemente eligen cero. El entrenamiento que elige una pendiente así en cada paso se llama descenso por subgradiente.

El libro añade una advertencia. Para curvas que se doblan en los dos sentidos, los subgradientes pueden no existir, y los matemáticos usan en su lugar una versión local, más flexible. Peor aún, dos programas que calculan exactamente la misma función pueden dar pendientes distintas en una esquina: el libro cita un caso en el que tres maneras de escribir la ReLU dan cero, uno y un medio en PyTorch en el mismo punto.



La misma función, pendientes distintas

El capítulo se cierra con la práctica. El libro recomienda que usted mismo construya un sistema diminuto en modo inverso, y señala micrograd, una pequeña versión didáctica; en cuanto las cadenas se ramifican y se juntan, hay que poner primero los pasos en el orden adecuado antes de recorrerlos hacia atrás. También sugiere escribir

una primitiva nueva con sus propias respuestas hacia atrás, en PyTorch o en JAX, y leer la guía de JAX sobre diferenciación automática para las pendientes de las pendientes y mucho más. Quedan fuera aquí las definiciones formales y los detalles de estos ejercicios; están en el apartado sobre la subdiferenciabilidad y en la sección final. Con esto queda montada toda la maquinaria que hay detrás del entrenamiento de cada asistente de chat y de cada aplicación de fotos: un error al final y un recorrido rápido hacia atrás que indica a cada perilla hacia dónde girar.

Para recordar

La ReLU tiene una esquina en cero, donde la pendiente ordinaria no existe. En la práctica el entrenamiento casi nunca cae exactamente ahí, y en teoría puede usarse cualquier pendiente de una recta que toque la esquina desde abajo, un subgradiente; para la ReLU, cualquier valor de cero a uno, normalmente cero. Aun así, las esquinas pueden hacer que programas distintos pero equivalentes den pendientes distintas. La mejor manera de entender todo esto es construir usted mismo un pequeño sistema en modo inverso.

Preguntas

1. ¿Por qué la esquina de la ReLU rara vez es un problema al entrenar en la práctica?

- a. Porque las bibliotecas eliminan la esquina antes de empezar el entrenamiento
 - b. Porque el entrenamiento empieza con números aleatorios y avanza en pasos ruidosos, así que caer exactamente en cero es prácticamente imposible
 - c. Porque la pendiente de la ReLU en cero es siempre uno
 - d. Porque la pasada hacia atrás se salta las capas ReLU
2. En un punto suave de una curva, ¿cuántos subgradientes hay?
- a. Exactamente uno, la derivada de siempre
 - b. Ninguno, los subgradientes solo existen en las esquinas
 - c. Dos, uno por cada lado
 - d. Toda una gama de cero a uno
3. ¿Qué pendientes son subgradientes válidos de la ReLU en cero?
- a. Solo cero
 - b. Solo uno
 - c. Cualquier valor de cero a uno, y las bibliotecas suelen elegir cero
 - d. Cualquier valor, ya que allí la pendiente no está definida
4. ¿Qué muestra el ejemplo del libro de las tres maneras de escribir la ReLU?
- a. Que dos de las tres maneras calculan una función distinta
 - b. Que PyTorch no puede derivar la ReLU en absoluto
 - c. Que el descenso por subgradiente nunca funciona con la ReLU
 - d. Que programas que calculan la misma función pueden dar pendientes distintas en una esquina

Respuestas

1. **b)** El entrenamiento empieza con números aleatorios y avanza en pasos pequeños y ruidosos, así que la probabilidad de una entrada exactamente igual a cero es prácticamente nula, y en todos los demás puntos la pendiente está definida.
2. **a)** Donde una curva es suave solo hay una recta que la toca desde abajo, la derivada de siempre; solo en una esquina hay muchas.
3. **c)** Para la ReLU en cero, cualquier pendiente de cero a uno da una recta que toca la esquina y queda por debajo de la curva, y la mayoría de las bibliotecas eligen cero.
4. **d)** Tres maneras de escribir la misma ReLU dan cero, uno y un medio en PyTorch en el mismo punto: las esquinas pueden hacer que programas equivalentes den pendientes distintas.

Ejemplos resueltos

La misma función, pendientes distintas Una segunda ReLU se escribe así: la ReLU de menos s , más s . Una tercera es la mitad de la suma de la ReLU ordinaria y la segunda. PyTorch usa la pendiente 0 para la ReLU ordinaria en cero. ¿Qué pendiente da la pasada hacia atrás en cero para la segunda y para la tercera?

La segunda función es igual a la ReLU en todas partes: para s positivo da $0 + s = s$, para s negativo da menos s más $s = 0$.

Su pasada hacia atrás en cero suma la pendiente de la parte ReLU, tomada como 0, y la pendiente de más s , que es 1: en total, 1.

La tercera es la mitad de (ReLU + segunda ReLU), así que su pendiente es la mitad de $(0 + 1)$.

Así, las tres funciones iguales dan 0, 1 y 0,5 en el mismo punto.

Respuesta: 1 para la segunda y 0,5 para la tercera, frente a 0 para la ReLU ordinaria

Ejercicios

1. ¿Es una pendiente de 0,3 un subgradiente válido de la ReLU en cero? ¿Y una pendiente de 1,5?

Pista: Imagine una recta por la esquina con esa pendiente: ¿queda por debajo del suelo y de la rampa?

Respuesta: 0,3 es válida, porque está entre 0 y 1 y su recta queda por debajo del suelo y de la rampa. 1,5 no lo es: para entradas positivas su recta sube más rápido que la rampa y pasa por encima de la curva.

2. Explique con sus propias palabras por qué un sistema casero en modo inverso debe poner los pasos en el orden adecuado antes de recorrerlos hacia atrás cuando la cadena se ramifica y se junta.

Pista: Piense en una magnitud que usan dos pasos posteriores: ¿cuándo está completa la fila que le vuelve?

Respuesta: Cuando las cadenas se ramifican y se juntan, una magnitud puede alimentar a varios pasos posteriores, y su fila hacia atrás solo está completa cuando todos ellos le han devuelto su contribución. Ordenar antes los pasos hace que el recorrido hacia atrás visite cada magnitud solo después de todo lo que depende de ella, así que no se toma ningún camino equivocado.