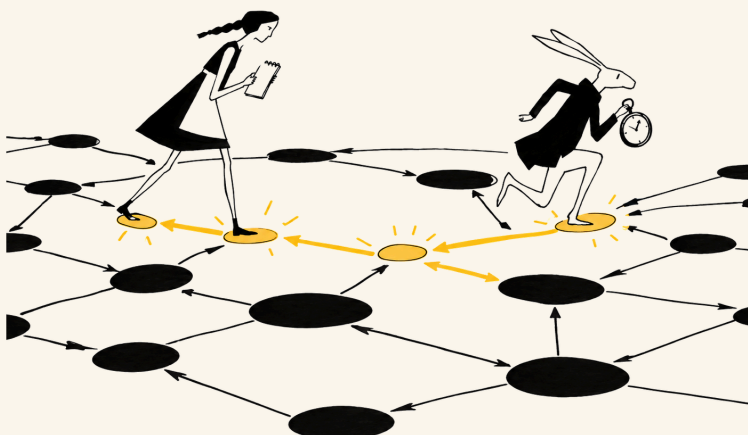


Alice's Adventures in a Differentiable Wonderland



Simone Scardapane · 2024

Illustrated audio lectures

Lecturized

Alice's Adventures in a Differentiable Wonderland

Illustrated audio lectures

Simone Scardapane · 2024

Retold as illustrated lectures by Lecturized
lecturized.com

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

Contents

A Program Made of Small Steps

Flattening the Tables and Stating the Task

Three Ways to Get a Derivative

Turning the Chain Around

Saving Memory and Asking Only for Products

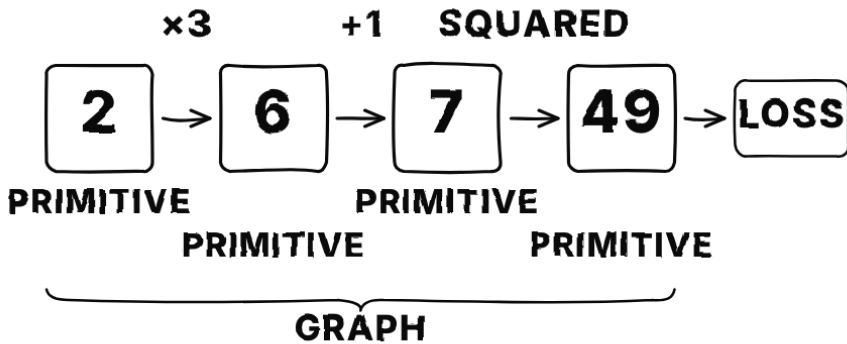
The Backward Step of a Layer

Frameworks, and Why the ReLU Wins

The Corner of the ReLU

A Program Made of Small Steps

Open a spreadsheet and build a tiny chain of cells. In the first cell type a number, say 2; in the second, a formula that triples the first; in the third, one that adds 1 to the second; in the last, one that squares the third. The last cell shows 49, because 2 times 3 is 6, plus 1 is 7, and 7 squared is 49. Now ask a sharper question: if you nudge the first cell up a little, how fast does the last cell move? You could try it by hand, typing 2.01 and watching the result, but the spreadsheet already knows every step it took, and each step on its own is easy to differentiate.

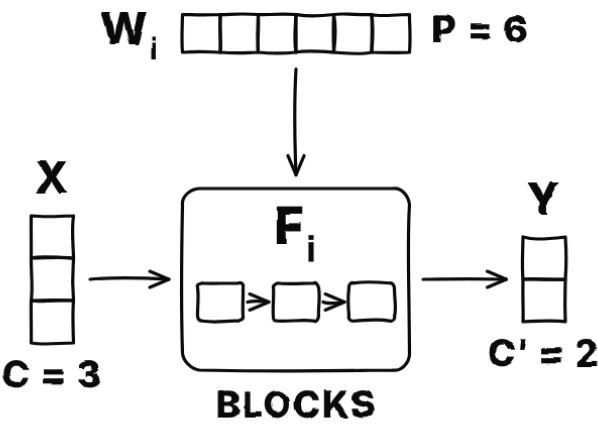


That is exactly the problem this chapter of the book takes up. When a chat assistant is trained, a huge program of such steps turns a sentence into a single number, the loss, which says how wrong the answer was. To improve the model, the computer needs the gradient: how the loss changes when each of millions of adjustable numbers is nudged. The book calls the task of getting these gradients efficiently, for any program, automatic differentiation, shortened to AD. The method it builds towards has two names: backpropagation among people who work on neural networks, and reverse-mode automatic differentiation among computer scientists.

The author asks us to picture a computational graph: the set of small, atomic operations that you get by actually running the program, like the cells of our spreadsheet. Each of these operations is called a primitive. For brevity the book follows a single chain of primitives, one after another, and notes that everything carries over to graphs that branch, as long as they have no loops, and even to more general ones.

At first the problem looks already solved. The chain rule of Jacobians, met earlier in the book, says that the derivative of a composition of functions is just the product of their Jacobian matrices, one per step. A Jacobian is a table of partial derivatives: one row for each output number, one column for each input number, every entry saying how much that output moves when that input is nudged. So why a whole chapter? Because doing this product cleverly, without wasting time and memory, is the real challenge, and the algorithm that meets it is a cornerstone of neural networks and of differentiable programming in general. The author adds that understanding it also explains how frameworks such as TensorFlow, PyTorch and JAX are designed, and why they differ.

Here is how the book sets up its building blocks. Each primitive is written as $\mathbf{y} = f_i(\mathbf{x}, \mathbf{w}_i)$. In words: the i -th primitive takes an input vector $\mathbf{x}$ and a vector of adjustable numbers $\mathbf{w}_i$, its weights, and produces an output vector $\mathbf{y}$. The shapes are written with a tilde: $\mathbf{x} \sim (c_i)$, $\mathbf{w}_i \sim (p_i)$, and $\mathbf{y} \sim (c'_i)$. Where does the form come from? Think of a linear projection in a layer of a network: the input comes from the step before, the weights belong to the step itself, and the result is passed on. With numbers: a small layer that takes 3 numbers in and gives 2 numbers out, and has 6 weights, has c equal to 3, c' equal to 2, and p equal to 6.

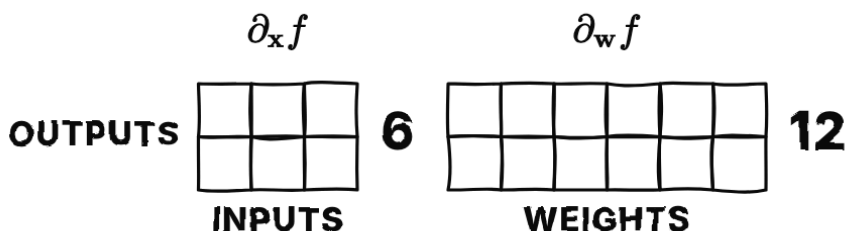


One Primitive: Input, Weights, Output

The definition is deliberately loose. A primitive may be a single matrix multiplication, a whole layer of the kind the previous chapter built,

such as a fully-connected layer with its activation function, or an entire block or model. Blocks inside blocks: this recursive composability is a key property of programming, and the book keeps it.

The only thing assumed about each primitive is that we know how to compute its partial derivatives with respect to its two arguments. The first is the input Jacobian, $\partial_{\mathbf{x}}[f(\mathbf{x}, \mathbf{w})] \sim (c', c)$: how each output moves when each input is nudged. The second is the weight Jacobian, $\partial_{\mathbf{w}}[f(\mathbf{x}, \mathbf{w})] \sim (c', p)$: how each output moves when each weight is nudged. The shapes follow from the rule of one row per output and one column per thing being nudged. For our small layer, the input Jacobian is a table of 2 rows and 3 columns, 6 numbers, and the weight Jacobian has 2 rows and 6 columns, 12 numbers.



The Two Jacobians of a Layer

These assumptions are reasonable, the author says, because the book restricts itself to differentiable models. A primitive that is continuous but has a few points without a derivative, such as the ReLU, which keeps positive numbers and sets negative ones to zero, is fitted in later with a tool called the subgradient. Operations that are not differentiable at all, such as drawing a random sample or cutting at a threshold, can also be handled by relaxing their gradient or using an equivalent estimator; the author leaves that for the next volume.

Remember

Training a network needs the gradient of one number, the loss, with respect to every weight, and automatic differentiation is the task of computing all of them efficiently for any program. A program, once run, is a chain of small primitives, each written $y = f_i(\mathbf{x}, \mathbf{w}_i)$: an input, its own weights, an output. For each primitive we only need two tables of partial derivatives, the input Jacobian of shape c' by c and the weight Jacobian of shape c' by p . The chain rule already says the answer is a product of Jacobians; the whole difficulty, and the reason backpropagation matters, lies in computing that product cheaply.

Questions

1. The chain rule already says that the derivative of a composition is a product of Jacobians. Why does the book still need a whole chapter on automatic differentiation?
 - a. Because the chain rule does not hold for neural networks
 - b. Because computing that product cleverly, without wasting time and memory, is the real challenge
 - c. Because the Jacobians of primitives are usually unknown
 - d. Because the loss is a vector and the chain rule needs a scalar
2. A primitive takes an input with c entries, has p weights and gives an output with c' entries. What shape does its weight Jacobian have?

- a. c rows and p columns
 - b. c' rows and c columns
 - c. c' rows and p columns
 - d. p rows and c' columns
3. What does the book assume we know about each primitive?
- a. How to compute its partial derivatives with respect to its input and its weights
 - b. Only how to compute its output, nothing about derivatives
 - c. Its full symbolic formula, so that a computer algebra system can work on it
 - d. That it is a single matrix multiplication
4. How does the book fit a primitive like the ReLU, which has a point without a derivative, into this framework?
- a. It drops the ReLU and uses only smooth functions
 - b. It postpones the ReLU to the next volume together with sampling
 - c. It uses a tool called the subgradient for the points without a derivative
 - d. It replaces the ReLU by a numerical approximation of its derivative

Answers

- 1. b)** The answer is in principle a product of Jacobians; the difficulty, and the reason backpropagation matters, is doing that product efficiently.
- 2. c)** A Jacobian has one row per output and one column per thing being nudged; for the weight Jacobian the nudged things are the p weights, so its shape is c' by p .

- 3. a)** The only assumption is that each primitive comes with its input Jacobian and its weight Jacobian.
- 4. c)** Continuous primitives with a few points of non-differentiability are fitted in with subgradients; only fully non-differentiable operations such as sampling are left for the next volume.

Worked examples

Sizing the two Jacobians of a layer A primitive $y = f(\mathbf{x}, \mathbf{w})$ takes an input of 3 numbers, has 6 weights and returns 2 numbers. How many entries do its input Jacobian and its weight Jacobian have?

Read off the sizes: $c = 3$ inputs, $p = 6$ weights, $c' = 2$ outputs.

The input Jacobian has one row per output and one column per input, so its shape is $(c', c) = (2, 3)$.

That makes $2 \times 3 = 6$ entries, each saying how one output moves when one input is nudged.

The weight Jacobian has one row per output and one column per weight, so its shape is $(c', p) = (2, 6)$, which is $2 \times 6 = 12$ entries.

Answer: The input Jacobian has 6 entries, the weight Jacobian 12.

Practice

1. A primitive takes 4 numbers in, has 12 weights and gives 3 numbers out. Write the shapes of its input Jacobian and its weight Jacobian.

Hint: One row per output; one column per input for the first, per weight for the second.

Answer: Input Jacobian $(3, 4)$, weight Jacobian $(3, 12)$: the outputs give the rows, the inputs or the weights give the columns.

2. Explain in your own words why the book is happy to call a single matrix multiplication, a whole layer, and an entire model all

'primitives'.

Hint: Think about what the method actually needs to know about each step.

Answer: The method only needs, for each step, its output and its two Jacobians, so it does not matter how big the step is. A block made of smaller blocks is still a step with an input, weights and an output; this recursive composability, a key property of programming, lets the same reasoning apply at every scale.

3. Training a chat assistant turns a sentence into one number, the loss. What does automatic differentiation have to deliver for that loss, and why is that the thing training needs?

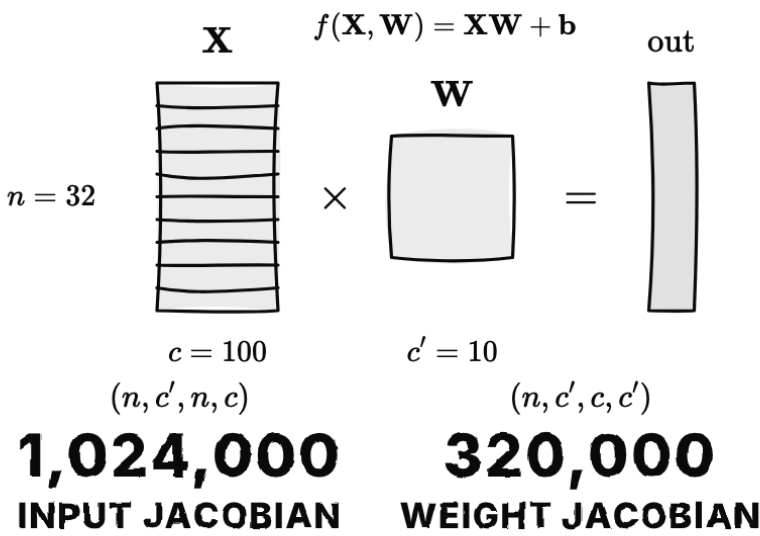
Hint: What must the computer know about each adjustable number before it can improve the model?

Answer: It must deliver the gradient of the loss with respect to every weight, for all weights at once and efficiently. Training changes each weight in the direction that lowers the loss, and the gradient is exactly what says how the loss moves when each weight is nudged.

Flattening the Tables and Stating the Task

Picture a spreadsheet with 32 rows and 100 columns: each row is one photo from your phone, each column one number the photo app has measured about it. The app does not look at one photo at a time; it takes the whole sheet, a mini-batch, in one go. Now ask how many partial derivatives describe a single layer working on that sheet. You might expect a few hundred. The honest count runs to about a million, and seeing why is the first lesson of this part.

BATCH THROUGH ONE LAYER

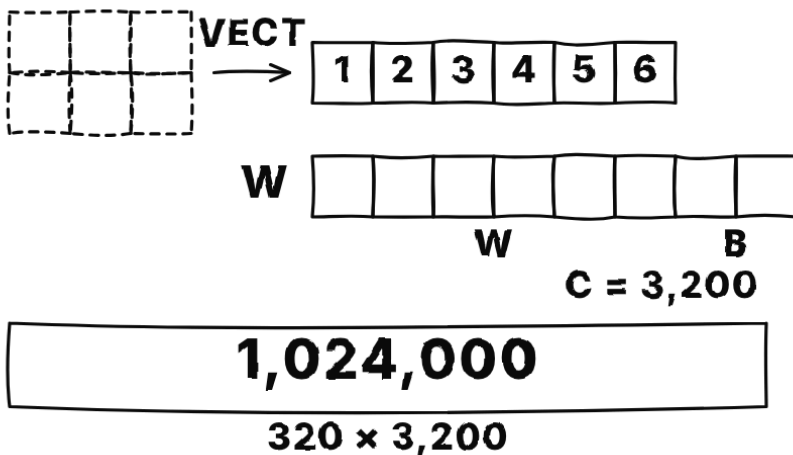


A Batch Through One Layer

So far the book has spoken of vectors, so that every Jacobian is a flat table, a matrix. Real primitives often take and give arrays of higher rank, that is, with more than two directions. The book's example is a fully-connected layer on a mini-batch: $f(\mathbf{X}, \mathbf{W}) = \mathbf{XW} + \mathbf{b}$. In words: the batch $\mathbf{X}$ is multiplied by the weight matrix $\mathbf{W}$, and the bias vector $\mathbf{b}$ is added to every row. The input $\mathbf{X}$ has shape (n, c) , with n examples of c numbers each; $\mathbf{W}$ has shape (c, c') and $\mathbf{b}$ has shape (c') , where c' , the number of outputs, is a hyper-parameter, a size we choose; the output has shape (n, c') .

Where do the Jacobians' shapes come from? A Jacobian pairs every output number with every number being nudged. The output has two directions, n and c' ; the input has two, n and c ; so the input Jacobian has shape (n, c', n, c) , an array of rank 4. The weight Jacobian, pairing the output with the matrix $\mathbf{W}$, has shape (n, c', c, c') , also rank 4. With the photo sheet, n is 32 and c is 100; take c' equal to 10. The input Jacobian then holds 32 times 10 times 32 times 100 entries, which is 1,024,000; the weight Jacobian holds 32 times 10 times 100 times 10, which is 320,000.

The book escapes the higher ranks with a simple trick: flattening. Read the array row after row and lay all its numbers out in one long vector, written $\mathbf{x} = \text{vect}(\mathbf{X})$. The weights become one vector too, the flattened matrix followed by the bias: $\mathbf{w} = [\text{vect}(\mathbf{W}); \mathbf{b}]$. Now the Jacobians are ordinary matrices again: the input Jacobian of shape (nc', nc) and the weight Jacobian of shape (nc', cc') . Check it with the photos: 320 rows times 3,200 columns is again 1,024,000, and 320 times 1,000 is again 320,000; nothing was lost, only rearranged.



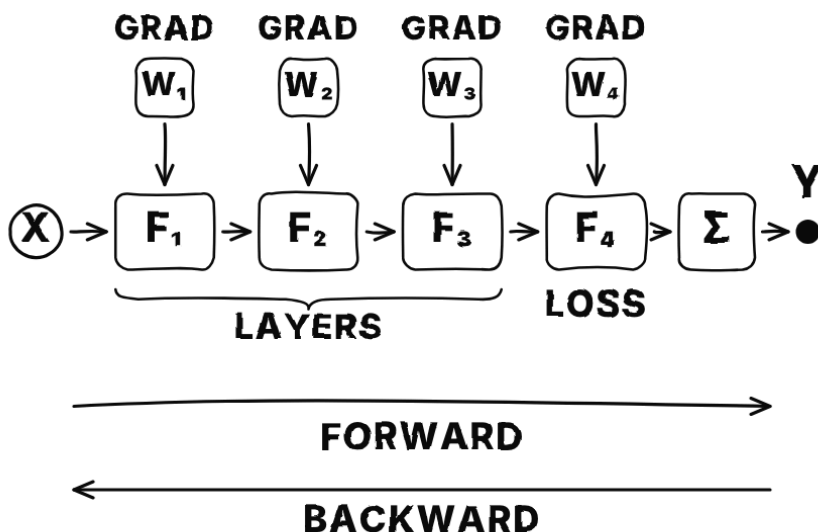
Flattening a Table

Two consequences matter for everything that follows. First, whenever the book says "the input size c ", it means the product of all the input's shapes, the mini-batch included; for our sheet that is 3,200, not 100. Second, a million entries for one modest layer is a warning: even when we know how to compute a Jacobian, we may not want to materialize it, that is, actually build it in memory. Remember that warning; the chapter will turn it into a method.

A last note on notation. Writing each primitive as a function of input and weights matches functional libraries such as JAX, where everything is a function. Object-oriented frameworks such as TensorFlow and PyTorch make a layer an object that keeps its parameters inside, with a method in place of the function call. That

style allows lazy initialization, creating the parameters only once the input's shape is known, but it adds a layer of abstraction between the book's formulas and working code, and the two styles also implement automatic differentiation differently.

Now the task can be stated. Take a program of l primitives in a row: $\mathbf{h}_1 = f_1(\mathbf{x}, \mathbf{w}_1)$, $\mathbf{h}_2 = f_2(\mathbf{h}_1, \mathbf{w}_2)$, and so on, up to $\mathbf{h}_l = f_l(\mathbf{h}_{l-1}, \mathbf{w}_l)$, and finally $y = \Sigma \mathbf{h}_l$. This listing is called the evaluation trace of the program. Roughly, the first $l - 1$ steps are the layers of the model, step l is the loss of each example, such as the cross-entropy, and the final sum adds the losses over the mini-batch. So the program always ends in a single number, a scalar, because numerical optimization needs one number to make smaller; the book calls the whole program $F(\mathbf{x})$. With three layers, l is 4: three layers, one loss step, then the sum.



The Evaluation Trace

The definition follows. Automatic differentiation is the task of computing, simultaneously and efficiently, all weight Jacobians of the program, knowing its graph and the input and weight Jacobians of every primitive: $AD(F(\mathbf{x})) = \{\partial_{\mathbf{w}_i} y\}$, $i = 1, \dots, l$. Because y is one number, each of these is as large as its weights: one entry per weight, telling how the loss moves when that weight is nudged. There are two families of algorithms, forward-mode and backward-mode, which differ only in the order in which the individual pieces are multiplied together. The book announces that the backward mode, called backpropagation for neural networks, is much more efficient here, and that the derivation extends to branching graphs and to weight sharing, where one set of weights is used by several layers, as in chapter 13.

Remember

Primitives often work on arrays of rank higher than two, so their Jacobians would have rank 4; flattening every array into one long vector turns them back into ordinary matrices. From now on "the input size" means the product of all input shapes, mini-batch included, and the numbers grow fast: about a million entries for one small layer on 32 photos. That size is why we may not want to build Jacobians in memory at all, even when we can compute them. A program is an evaluation trace of l primitives ending in a sum, so its output y is always a single number. Automatic differentiation means computing all the weight gradients of y at once and cheaply; forward and backward mode differ only in the order of the multiplications, and backward wins.

Questions

1. Why does the book flatten $\mathbf{X}$ into $\text{vect}(\mathbf{X})$ and the weights into $[\text{vect}(\mathbf{W}); \mathbf{b}]$?
 - a. To reduce the number of partial derivatives the layer has
 - b. To turn rank-4 Jacobians back into ordinary matrices without losing any entry
 - c. Because the mini-batch dimension can then be ignored
 - d. Because functional libraries cannot store matrices

2. A layer works on a mini-batch of 32 photos with 100 numbers each. When the book speaks of 'the input size c ', what is c here?
 - a. 100, the numbers per photo
 - b. 32, the size of the mini-batch
 - c. 3,200, the product of all input shapes
 - d. 132, the sum of the two shapes
3. Why does the program in the evaluation trace always end in a single number y ?
 - a. Because each layer reduces its input to one number
 - b. Because numerical optimization needs one number to make smaller, so the per-example losses are summed
 - c. Because the Jacobians could not be flattened otherwise
 - d. Because forward-mode differentiation only works with scalars
4. What separates forward-mode from backward-mode automatic differentiation, as the book announces them?
 - a. The order in which the individual pieces are multiplied together
 - b. Forward mode uses numerical differentiation, backward mode symbolic
 - c. Forward mode needs the input Jacobians, backward mode only the weight Jacobians
 - d. Backward mode works only for object-oriented frameworks

Answers

1. **b)** Flattening only rearranges the numbers: the rank-4 arrays become matrices of shape (nc', nc) and (nc', cc') with exactly the same number of

entries.

2. **c)** The input size means the product of all the input's shapes, mini-batch included: 32 times 100 is 3,200.
3. **b)** The last steps compute a loss per example and then add them over the mini-batch, giving the scalar that training makes smaller.
4. **a)** Both compute the same weight gradients; they differ only in the order of composition, and the backward mode turns out much more efficient here.

Worked examples

Counting a layer's Jacobian entries A fully-connected layer $f(\mathbf{X}, \mathbf{W}) = \mathbf{X}\mathbf{W} + \mathbf{b}$ works on a mini-batch of $n = 8$ examples with $c = 4$ numbers each and gives $c' = 3$ outputs per example. How many entries does its input Jacobian have, and what shape does it take after flattening?

Before flattening, the input Jacobian pairs every output number with every input number, so its shape is $(n, c', n, c) = (8, 3, 8, 4)$.

Its entries: $8 \times 3 \times 8 \times 4 = 768$.

After flattening, the output is a vector of $n \cdot c' = 24$ numbers and the input one of $n \cdot c = 32$ numbers, so the Jacobian is a matrix of shape $(24, 32)$.

Check: $24 \times 32 = 768$, the same count, because flattening only rearranges the numbers.

Answer: 768 entries; flattened shape $(24, 32)$.

Practice

1. A layer works on $n = 16$ examples with $c = 50$ numbers each and gives $c' = 5$ outputs per example. Give the input size in the book's sense, and the flattened shapes of the input Jacobian and the weight Jacobian (with $\mathbf{W}$ of shape (c, c')).

Hint: The input size is the product of all input shapes; the flattened Jacobians are (nc', nc) and (nc', cc') .

Answer: Input size $16 \times 50 = 800$. Input Jacobian $(80, 800)$, that is 64,000 entries; weight Jacobian $(80, 250)$, that is 20,000 entries.

2. A model has 5 layers, followed by a per-example loss and a sum over the mini-batch. Write how its evaluation trace ends and say what l is.

Hint: Count the primitives before the final sum: the layers and the loss step.

Answer: The trace runs $\mathbf{h}_1 = f_1(\mathbf{x}, \mathbf{w}_1)$ up to $\mathbf{h}_6 = f_6(\mathbf{h}_5, \mathbf{w}_6)$, the sixth step being the loss, and ends with $y = \Sigma \mathbf{h}_6$. So l is 6, and y is a single number.

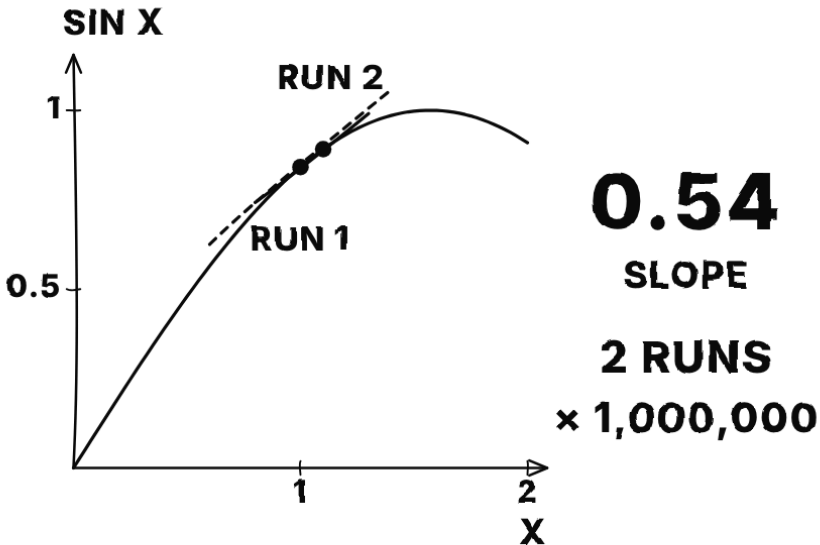
3. Explain in your own words why the book warns that we may not want to materialize the Jacobians even when we know how to compute them.

Hint: Think of the count you get for one modest layer on a mini-batch.

Answer: Once the input size includes the mini-batch, the flattened Jacobians become huge: about a million entries for one small layer on 32 photos. Building such tables in memory for every layer would waste space, so the method should get the gradients without ever storing them whole.

Three Ways to Get a Derivative

Take a pocket calculator and the function sine. Work out the sine of 1, then the sine of 1.001, subtract, and divide by 0.001. You get about 0.54, and the cosine of 1, the true derivative, is 0.5403. It works, and it took two runs of the function to get one derivative. Keep those two runs in mind, because they decide the fate of this first method.

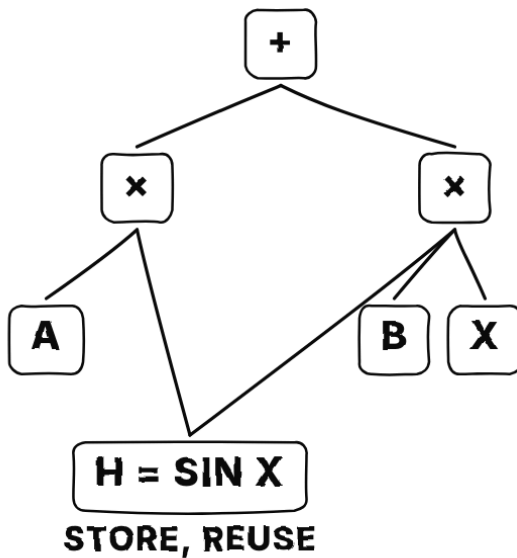


A Slope From Two Runs

What you just did is numerical differentiation: applying the definition of the derivative directly, with a small nudge, to get an approximation. The book points out its cost: in a naive implementation, every single

number you want to differentiate needs two calls of the function. A model with a million weights would need two million runs of the whole program for one gradient, and a chat assistant has far more weights than that. So the author keeps numerical differentiation for one job only: checking, now and then, that a faster implementation gives the right numbers.

The second way is symbolic differentiation. Take the book's small function $f(x) = a \sin(x) + b x \sin(x)$. A symbolic engine, a program for algebra, can work out the whole formula of the derivative in advance, and the book shows this with the Python library SymPy: you declare x , a and b as symbols, type the function, ask for its derivative, and it prints the result. That result is $f'(x) = a \cos(x) + b x \cos(x) + b \sin(x)$. Where does it come from? The first term turns sine into cosine; the second is a product, x times sine, so it gives two terms, one for each factor. Check it with numbers: take a equal to 2, b equal to 3, and x equal to $\pi/2$, where the sine is 1 and the cosine 0; the derivative is 2 times 0, plus 3 times $\pi/2$ times 0, plus 3 times 1, which is 3.

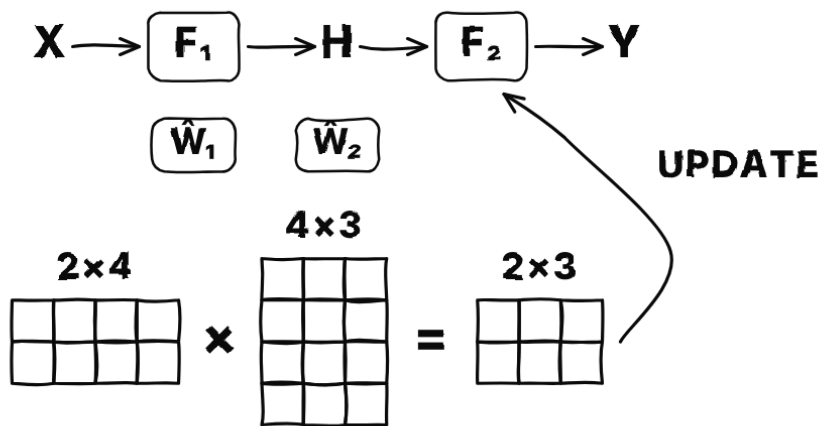


NP-COMPLETE

The Same Sine Computed Twice

Look at the formula again and you see repetition: $\sin(x)$ appears twice in the function, $\cos(x)$ twice in the derivative. A sensible program computes $h = \sin(x)$ once, stores it, and reuses it, also in the computation of the derivative, and the same for the cosine. That sounds like tidying up, but the author warns it is less trivial than it appears: finding the best possible way to compute a Jacobian, with no unnecessary step at all, is a problem called optimal Jacobian accumulation, and it is NP-complete, which means no fast general method for it is known. The good news is that a program has a structure we can exploit. Automatic differentiation uses it to be far better than the plain symbolic approach; the book adds that it is in fact equivalent to a symbolic approach that is allowed to store and reuse such intermediate pieces.

Now the third way, starting from the chain rule. Take two primitives in a row, $\mathbf{h} = f_1(\mathbf{x})$, $\mathbf{y} = f_2(\mathbf{h})$. The chain rule of Jacobians says $\partial_{\mathbf{x}}\mathbf{y} = \partial_{\mathbf{h}}\mathbf{y} \cdot \partial_{\mathbf{x}}\mathbf{h}$. If $\mathbf{x}$, $\mathbf{h}$ and $\mathbf{y}$ have a , b and c entries, that is a c by b matrix times a b by a matrix, and the result is c by a , one row per output, one column per input, as it should be. With 3 inputs, 4 middle values and 2 outputs, a 2 by 4 table times a 4 by 3 table gives a 2 by 3 table. The book reads the rule as an update: once you have f_1 and its Jacobian, applying f_2 means multiplying what you have by f_2 's Jacobian, and you are up to date again.



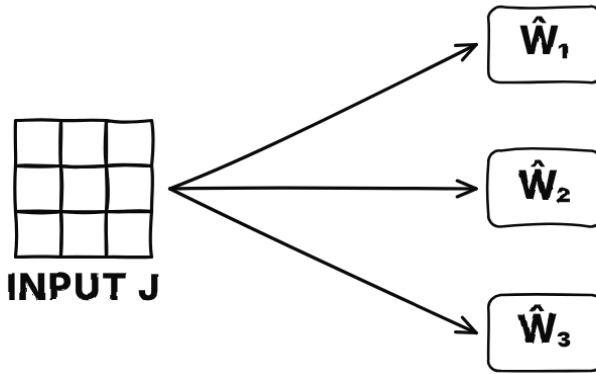
Chain Rule as a Table Product

That reading is already an algorithm, forward-mode automatic differentiation, F-AD for short. Each time a primitive runs, you start a new table for its weights, called a tangent, and you update all the

tangents you already hold. At the first step, $\mathbf{h}_1 = f_1(\mathbf{x}, \mathbf{w}_1)$, nothing is stored yet, so the tangent is just the weight Jacobian: $\hat{\mathbf{W}}_1 = \partial_{\mathbf{w}_1} \mathbf{h}_1$. At the second step, $\mathbf{h}_2 = f_2(\mathbf{h}_1, \mathbf{w}_2)$, the old tangent is multiplied by the input Jacobian of f_2 , $\hat{\mathbf{W}}_1 \leftarrow [\partial_{\mathbf{h}_1} \mathbf{h}_2] \hat{\mathbf{W}}_1$, and a new one starts, $\hat{\mathbf{W}}_2 = \partial_{\mathbf{w}_2} \mathbf{h}_2$.

Try it yourself

Try it with single numbers on a calculator. Let the input x be 2, the first step multiply by a weight w_1 equal to 3, and the second multiply by a weight w_2 equal to 5. After the first step the tangent is the derivative of 3 times 2 with respect to w_1 , which is x , so 2. At the second step the input Jacobian is w_2 , 5, so the tangent becomes 5 times 2, which is 10; the new tangent for w_2 is the value coming in, 6. Now check by hand: the output is w_2 times w_1 times x , its derivative with respect to w_1 is 5 times 2, which is 10, and with respect to w_2 it is 3 times 2, which is 6.



Forward Mode on a Calculator

For the i -th primitive in general, $\mathbf{h}_i = f_i(\mathbf{h}_{i-1}, \mathbf{w}_i)$, the rule is the same: $\hat{W}_j \leftarrow [\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i] \hat{W}_j$ for all $j < i$, and $\hat{W}_i = \partial_{\mathbf{w}_i} \mathbf{h}_i$. So the update uses the input Jacobian of the step and the new tangent its weight Jacobian. At step i there are $i - 1$ updates, one for each tangent in memory, and they all share the same input Jacobian; at the fourth step, three tangents are updated with one and the same table. The program's last operation is the sum, and its gradient will give the algorithm's output.

Remember

Numerical differentiation nudges each number and runs the function twice, which is fine for checking but hopeless for millions of weights. Symbolic differentiation writes out the derivative formula, but it repeats work, and finding the perfectly economical way to compute a Jacobian is an NP-complete problem. Automatic differentiation uses the structure of the program instead, like a symbolic method that may store and reuse intermediate values. Forward mode reads the chain rule as an update: at each step a new tangent starts from the weight Jacobian, and every older tangent is multiplied by the step's input Jacobian.

Questions

1. Why does the book keep numerical differentiation only for checking an implementation?
 - a. Because it gives the wrong sign for most derivatives
 - b. Because each number to differentiate needs two calls of the whole function, far too many for millions of weights
 - c. Because it cannot handle the sine function
 - d. Because it needs a symbolic engine to run
2. What is the trouble with simply writing out the derivative formula symbolically?
 - a. The formula is usually wrong for products

- b. Symbolic engines cannot differentiate sine
 - c. It repeats work, and finding the perfectly economical way to compute a Jacobian is NP-complete
 - d. It needs two function calls per number, like numerical differentiation
3. In forward mode, what happens to the tangents already in memory when the i-th primitive runs?
- a. Each is multiplied by the input Jacobian of the i-th primitive, one shared table for all of them
 - b. Each is multiplied by the weight Jacobian of the i-th primitive
 - c. They are discarded and recomputed from the start
 - d. They stay unchanged until the final sum

Answers

- 1. b)** In a naive implementation every scalar to differentiate costs two function calls, so a model with a million weights would need two million runs for one gradient.
- 2. c)** Pieces such as $\sin(x)$ and $\cos(x)$ appear several times; avoiding every unnecessary computation in general is the NP-complete problem of optimal Jacobian accumulation.
- 3. a)** The rule $\hat{W}_j \leftarrow [\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i] \hat{W}_j$ for all $j < i$ updates every older tangent with the same input Jacobian, while the new tangent starts from the weight Jacobian.

Worked examples

The book's symbolic example, at one point For the book's function $f(x) = a \sin(x) + b x \sin(x)$, the symbolic engine returns $f'(x) = a \cos(x) + b x \cos(x) + b \sin(x)$. Evaluate the derivative at $x = 0$.

At $x = 0$ the sine is 0 and the cosine is 1.

The first term gives a times 1, which is a .

The second term gives b times 0 times 1, which is 0, because of the factor x .

The third term gives b times 0, which is 0.

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

Forward mode through three steps A program multiplies its input $x = 1$ by $w_1 = 2$, then by $w_2 = 4$, then by $w_3 = 0.5$. Carry the tangents forward and give the derivative of the output with respect to each weight.

Step 1: $h_1 = w_1 x = 2$; the new tangent is the weight Jacobian, the derivative of $w_1 x$ with respect to w_1 , which is $x = 1$.

Step 2: $h_2 = w_2 h_1 = 8$; the input Jacobian is $w_2 = 4$, so the old tangent becomes $4 \times 1 = 4$; the new tangent is $h_1 = 2$.

Step 3: $h_3 = w_3 h_2 = 4$; the input Jacobian is $w_3 = 0.5$, so the tangents become $0.5 \times 4 = 2$ and $0.5 \times 2 = 1$; the new tangent is $h_2 = 8$.

Check: $h_3 = w_3 w_2 w_1 x$, whose derivatives are $w_3 w_2 x = 2$, $w_3 w_1 x = 1$ and $w_2 w_1 x = 8$.

Answer: 2 for w_1 , 1 for w_2 , 8 for w_3 .

Practice

1. A small model has 5,000 weights. How many calls of the program does naive numerical differentiation need for one full gradient?

Hint: Two calls for each number to be differentiated.

Answer: 10,000 calls: two for each of the 5,000 weights.

2. In $\mathbf{h} = f_1(\mathbf{x})$, $\mathbf{y} = f_2(\mathbf{h})$, the input has 5 entries, $\mathbf{h}$ has 7 and $\mathbf{y}$ has 1. Give the shapes of the two Jacobians in the chain rule and of their product.

Hint: One row per output, one column per input, for each step.

Answer: $\partial_{\mathbf{y}}\mathbf{h}$ is 1 by 7, $\partial_{\mathbf{x}}\mathbf{h}$ is 7 by 5, and their product $\partial_{\mathbf{x}}\mathbf{y}$ is 1 by 5.

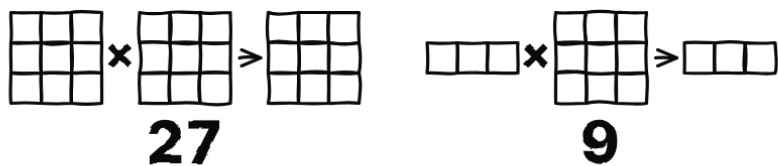
3. In forward mode, how many tangent updates happen at the sixth primitive, and which Jacobian do they share?

Hint: Count the tangents already in memory before the sixth step.

Answer: Five updates, one for each of the tangents of $\mathbf{w}_1$ to $\mathbf{w}_5$, all using the same input Jacobian of the sixth primitive; then the sixth tangent starts from its weight Jacobian.

Turning the Chain Around

Take a pencil and two small tables of numbers, each 3 by 3, and multiply them. Every entry of the result needs 3 multiplications, and there are 9 entries, so you make 27 multiplications. Now multiply a single row of 3 numbers by one of those tables. The result is a row of 3 entries, 3 multiplications each: 9 in all. A row times a table is three times cheaper here, and with tables of a thousand rows the gap becomes a thousandfold; that gap is the whole story of this part.



$\mathbf{1}^T$

1

1

×

$\hat{W}$

4

6

=

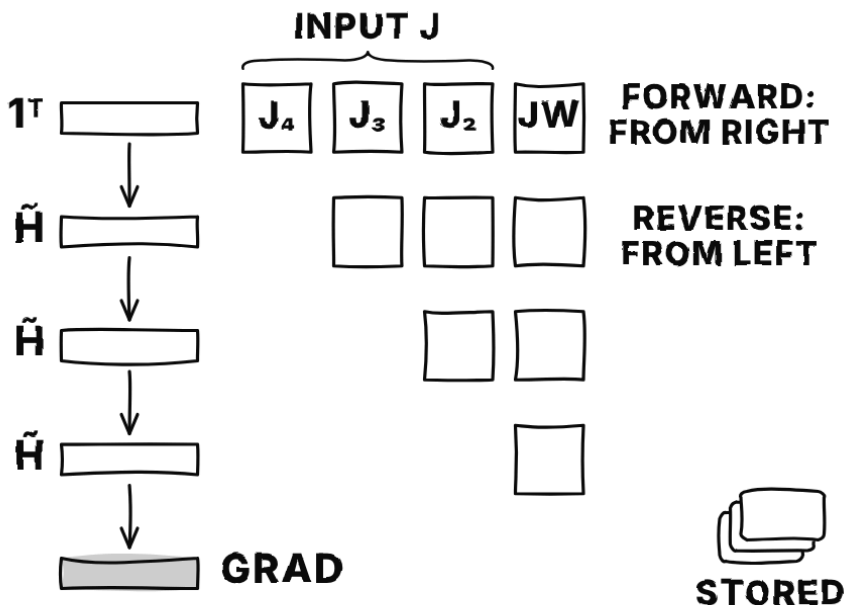
10

Table Times Table, Row Times Table

First, forward mode finishes its work. The program ends with the sum of $\mathbf{h}_i$, so the gradient for each weight comes from the last tangent: $\nabla_{\mathbf{w}_i} y = \mathbf{1}^T \hat{\mathbf{W}}_i$ for all i . In words: $\mathbf{1}$ is a vector filled with ones, and multiplying by it adds up the rows of the tangent, which is exactly what the final sum does to the outputs. With two outputs and a tangent whose rows are 4 and 6, the gradient is 4 plus 6, which is 10. A small print note in the book: this gives a row rather than the column it defined gradients to be; the author sets that aside until later in the chapter.

How good is forward mode? On the good side, every update can be interleaved with the original program, step by step, so the memory it needs is roughly what the program itself needs. On the bad side, its central move, updating a tangent, multiplies two matrices, of shapes (c'_i, c_i) and (c_i, p_j) , where p_j is the number of weights in $\mathbf{w}_j$. If inputs and outputs are both shaped (n, d) , with n examples in the mini-batch and d features each, the cost is $\mathcal{O}(n^2 d^2 p_j)$: it grows with the square of the batch and the square of the features. With 32 photos, 100 features and 10,000 weights, that is 1,024 times 10,000 times 10,000, about a hundred billion multiplications, for one update of one tangent. For images, with many more features, it quickly becomes unfeasible.

The way out is hidden in the last step of forward mode, which is not a matrix times a matrix but a vector times a matrix, because the output is a single number. To use it, the book writes out the gradient for one weight in full: $\nabla_{\mathbf{w}_i} y = \mathbf{1}^T [\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i] \cdots [\partial_{\mathbf{h}_i} \mathbf{h}_{i+1}] [\partial_{\mathbf{w}_i} \mathbf{h}_i]$. Notation apart, it is just a long row of multiplications: the vector of ones, then a chain of input Jacobians, which the book prints in red, and at the far right the weight Jacobian of step i , in green. The red part gets a name of its own: $\tilde{\mathbf{h}}_i = \mathbf{1}^T \prod_{j=i+1}^l \partial_{\mathbf{h}_{j-1}} \mathbf{h}_j$.



Multiply From the Left

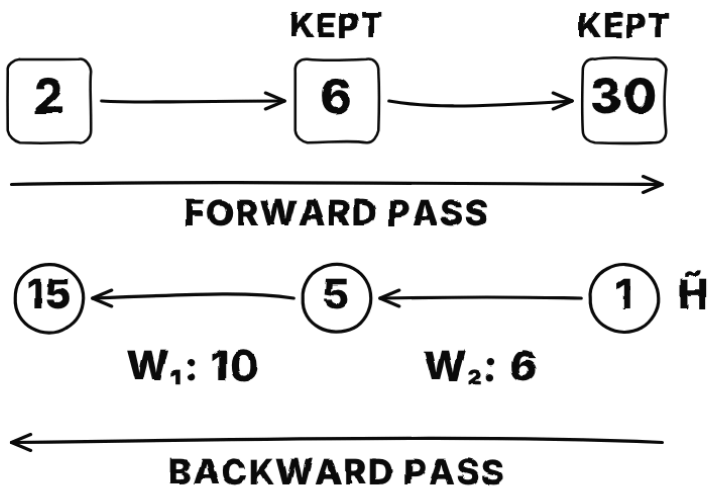
Matrix multiplication is associative: you may bracket the product any way you like and the result is the same. Forward mode multiplied from right to left, the order in which the program ran, always holding a matrix. Multiply instead from the left, and two things happen. The leftmost piece is a vector times a matrix, cheap, and its result is again a vector, so the cheapness never runs out. And $\tilde{\mathbf{h}}_i$ can be built recursively, starting from the last step and multiplying by the input Jacobians in reverse order.

This gives reverse-mode automatic differentiation, R-AD, in three steps. First, run the entire program and store every intermediate output. Second, start with the vector $\tilde{\mathbf{h}} = \mathbf{1}^T$, the leftmost term. Third, walk backwards, i going from l down to 1 , and at each step do two

things: get the gradient for that step's weights, $\partial_{\mathbf{w}_i} y = \tilde{\mathbf{h}} [\partial_{\mathbf{w}_i} \mathbf{h}_i]$, and then pass the vector one step back, $\tilde{\mathbf{h}} \leftarrow \tilde{\mathbf{h}} [\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i]$. Both are a vector times a matrix.

Try it yourself

Take the calculator chain from the last part again: x is 2, multiplied by w_1 equal to 3, then by w_2 equal to 5. Run it forward and write down what you see: 6 after the first step, 30 after the second. Now start backwards with $\tilde{h}$ equal to 1. At step 2 the gradient for w_2 is 1 times the value that came in, 6; then $\tilde{h}$ becomes 1 times w_1 , which is 5. At step 1 the gradient for w_1 is 5 times x , which is 10; you get the same 10 and 6 as forward mode, and the last $\tilde{h}$, 15, is the derivative of the output with respect to x .



Backwards on the Calculator

Those three steps form a program roughly mirroring the original, which the book calls the dual or reverse program. The vectors $\tilde{\mathbf{h}}$ are called adjoints: one after another they hold the gradient of the output with respect to each $\mathbf{h}_i$, whereas forward mode's tangents held the gradients of each $\mathbf{h}_i$ with respect to the weights. In the language of neural networks, the original or primal program is the forward pass, which is not the same thing as forward mode, and the reverse program is the backward pass. Unlike forward mode, the whole forward pass must finish before the backward pass can start, and frameworks need special machinery to store all the intermediate outputs, each in its own way. Training a photo app works exactly like this: the photos go through the network, the loss is computed, and then the gradients flow back.

Remember

Forward mode ends with $\nabla_{\mathbf{w}_i} y = \mathbf{1}^T \hat{\mathbf{W}}_i$ and uses little memory, but each tangent update multiplies two matrices, at a cost like $n^2 d^2 p_j$ that explodes for images. Written out, each gradient is a row of multiplications starting with the vector of ones; since the order of multiplying is free, starting from the left keeps every product a vector times a matrix. Reverse mode runs the program first, storing every intermediate value, then walks backwards carrying the adjoint $\tilde{\mathbf{h}}$, giving one weight gradient per step. The forward pass must finish before the backward pass starts, which is why everything in between has to be kept.

Questions

1. Why does forward mode become unfeasible for inputs such as images, even though it needs little memory?
 - a. Because it must store every intermediate output of the program
 - b. Because each tangent update multiplies two matrices, with a cost growing like $n^2 d^2 p_j$
 - c. Because it cannot be interleaved with the original program
 - d. Because images have no weight Jacobian
2. What makes it possible to compute the same product of Jacobians from the left instead of from the right?

- a. Matrix multiplication is associative, so the brackets can be placed freely
 - b. Jacobians are always square matrices
 - c. The input Jacobians are all equal to each other
 - d. The weight Jacobian is a vector of ones
3. Why must reverse mode run the whole program before the backward pass can start?
- a. Because the backward pass needs the loss value only at the very end
 - b. Because the backward pass starts from the output and needs the stored intermediate outputs of every step
 - c. Because the weights change during the forward pass
 - d. Because the vector of ones is only known after the program has run
4. What does an adjoint $\mathbf{h}_i$ hold, compared with a tangent of forward mode?
- a. The gradient of $\mathbf{h}_i$ with respect to the weights, like a tangent
 - b. The value of $\mathbf{h}_i$ itself, stored during the forward pass
 - c. The gradient of the output with respect to $\mathbf{h}_i$, whereas a tangent holds the gradient of $\mathbf{h}_i$ with respect to the weights
 - d. The weight Jacobian of step i

Answers

1. **b)** The core update multiplies matrices of shapes (c'_i, c_i) and (c_i, p_j) ; with inputs and outputs shaped (n, d) that costs on the order of $n^2 d^2 p_j$, quadratic in batch size and features.

- 2. a)** Because matrix multiplication is associative, the long product can be done in any order; starting from the vector of ones keeps every step a vector times a matrix.
- 3. b)** The adjoint starts at the output and walks back through every step, using Jacobians that depend on the intermediate values, so all of them must have been computed and stored first.
- 4. c)** Adjoint stores, step by step, the gradients of the output with respect to each intermediate value; tangents went the other way, from weights to intermediate values.

Worked examples

A backward pass through three steps A program multiplies $x = 1$ by $w_1 = 2$, then by $w_2 = 4$, then by $w_3 = 0.5$, and its output is the result. Use reverse mode to get the gradient for each weight.

Forward pass, storing everything: $h_1 = 2$, $h_2 = 8$, $h_3 = 4$.

Start the adjoint at $\tilde{h} = 1$.

Step 3: the gradient for w_3 is $\tilde{h}$ times the value that came in, $1 \times 8 = 8$; then $\tilde{h}$ becomes $1 \times w_3 = 0.5$.

Step 2: the gradient for w_2 is $0.5 \times h_1 = 1$; then $\tilde{h}$ becomes $0.5 \times w_2 = 2$.

Step 1: the gradient for w_1 is $2 \times x = 2$; the last $\tilde{h}$, $2 \times w_1 = 4$, is the derivative of the output with respect to x .

Answer: 8 for w_3 , 1 for w_2 , 2 for w_1 .

Practice

1. A tangent has two rows, $(1, 2)$ and $(3, 4)$. What does $\mathbf{1}^T$ times this tangent give?

Hint: Multiplying by a vector of ones adds up the rows.

Answer: The row $(4, 6)$: $1 + 3$ and $2 + 4$.

2. Forward mode's update costs on the order of $n^2 d^2 p_j$ multiplications. Estimate it for $n = 10$ examples, $d = 20$ features and $p_j = 50$ weights, then say what happens when the batch doubles.

Hint: Square n and d , then multiply by p_j .

Answer: $100 \times 400 \times 50 = 2,000,000$ multiplications. Doubling the batch multiplies the cost by four, because it grows with n squared.

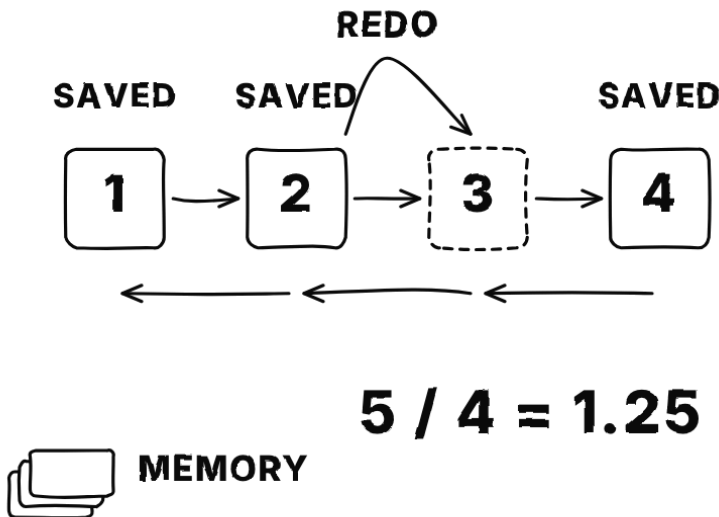
3. Explain why carrying a vector from the left is cheaper than carrying a matrix from the right, for a program whose output is a single number.

Hint: Compare a vector times a table with a table times a table.

Answer: Because the output is one number, the leftmost term is a vector, and a vector times a matrix gives another vector; every step stays a vector-matrix product. Going from the right, each step multiplies two matrices, which for tables of a thousand rows costs about a thousand times more.

Saving Memory and Asking Only for Products

Think of a long video game level with save points. If you could save after every room, a mistake would cost you nothing, but your memory card would fill up quickly. If you save only every few rooms, the card stays light, and after a mistake you replay a few rooms from the last save. Players make this trade without thinking about it. Reverse mode has to make exactly the same trade, and the book's solution even carries the gamer's name: checkpoints.



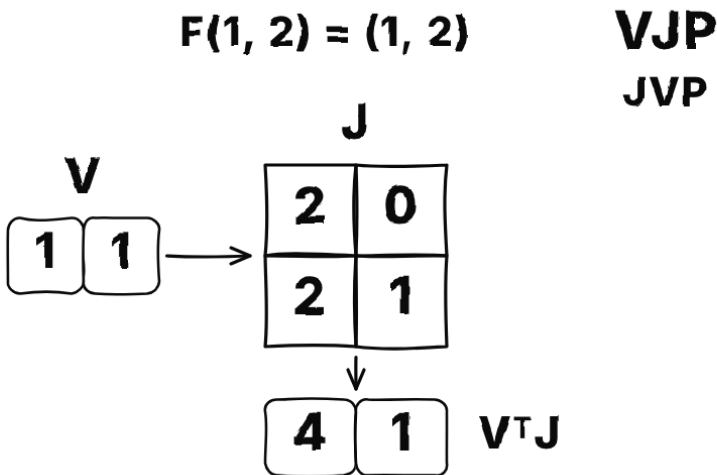
Checkpoints Along Four Blocks

First the good news, in numbers. Reverse mode is far more efficient than forward mode, because both moves of its backward step are a vector times a matrix, and that cost grows only linearly with every size involved. Double the batch and the work doubles, where forward mode's quadruples. The price is memory: every intermediate value of the primal program has to be kept for the backward pass, and the book says it must be stored with a suitable strategy, even on disk. For a network trained on high-resolution photos, those stored values can take far more room than the weights themselves.

Gradient checkpointing improves the trade by spending some extra computation to save part of the memory. Only a few intermediate outputs are stored, the checkpoints, and the others are recomputed during the backward pass when they are needed. The book's figure shows it in four panels, with a model of four blocks drawn in a row. In the first, a forward pass runs, but only the outputs of the first, second and fourth blocks are kept. In the second, the backward pass, drawn as red arrows going right to left, stops at the third block, because its values were not stored. In the third, a second forward pass starts from the nearest checkpoint, the second block, and computes the third block's values again; in the fourth, the backward pass completes. Count the work: four blocks forward, plus one block recomputed, is five block computations instead of four, which is 1.25 times as much, the figure the book gives. The fewer checkpoints you keep, the more the backward pass costs.

Now look again at the third step of reverse mode. The only operation it ever needs is a row vector $\mathbf{v}$ multiplied by a Jacobian of f , either the input Jacobian or the weight Jacobian. These two operations are called the vector-Jacobian products of f , VJPs for short. The book defines one formally, with a transpose to put the vector the right way

round: for $\mathbf{y} = f(\mathbf{x})$, with $\mathbf{x} \sim (c)$ and $\mathbf{y} \sim (c')$, it is $\text{vjp}_f(\mathbf{v}) = \mathbf{v}^\top \partial f(\mathbf{x})$, where $\mathbf{v}$ has c' entries, one per output. In words: you hand the function a vector the size of its output, and it hands you back a vector the size of its input, as if the Jacobian had been used, without ever showing you the Jacobian. A function with several arguments has one VJP per argument.

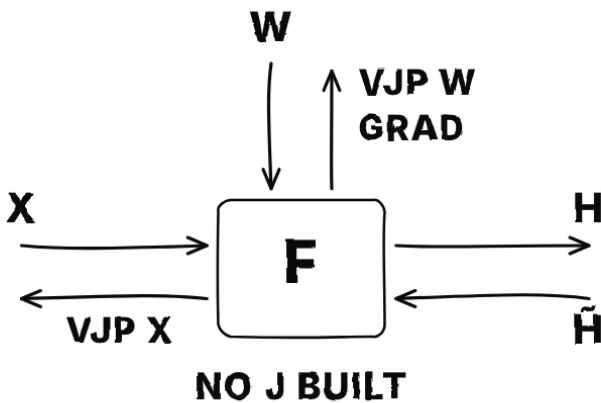


A Vector Times the Jacobian

A small example to see the numbers. Let f take $\mathbf{x} = (1, 2)$ to $(x_1^2, x_1 x_2)$, that is, to $(1, 2)$. Its Jacobian has rows $(2x_1, 0)$ and (x_2, x_1) , so $(2, 0)$ and $(2, 1)$. With $\mathbf{v} = (1, 1)$, the VJP is 1 times the first row plus 1 times the second: $(4, 1)$. By contrast, the book notes in passing that forward mode can be written entirely with the transposed operation,

the Jacobian-vector product, JVP, which for a single output is the directional derivative met earlier.

A primitive has two arguments, so it has two VJPs: $\text{vjp_f, x}(\mathbf{v}) = \mathbf{v}^\top \partial_{\mathbf{x}} f(\mathbf{x}, \mathbf{w})$ and $\text{vjp_f, w}(\mathbf{v}) = \mathbf{v}^\top \partial_{\mathbf{w}} f(\mathbf{x}, \mathbf{w})$. The two moves of the backward step become two calls with the adjoint: $\partial_{\mathbf{w}} y = \text{vjp_f, w}(\mathbf{h})$, and $\mathbf{h} \leftarrow \text{vjp_f, x}(\mathbf{h})$. So a whole automatic differentiation system can be built by choosing a set of primitives and giving each of them its two VJPs, without materializing a Jacobian in memory at any point. The book's second figure draws a primitive as a box: in the forward pass the input and weights go in and the output comes out; in the backward pass the adjoint comes in from the right, and two calls, one per VJP, send one vector back towards the input and one to the weights. This is what happens, step after step, when a chat assistant's training run calls its backward pass.



$E_1 \rightarrow (2, 0)$
 $E_2 \rightarrow (2, 1)$

A Primitive With Two VJPs

Nothing is lost by working this way. Call the input VJP with the basis vectors $\mathbf{e}_1, \dots, \mathbf{e}_n$, each a vector with a 1 in one place and 0 elsewhere, and you get the Jacobian back one row at a time: $\partial_x f(\mathbf{x}, \mathbf{w}) = [\text{vjp_f, x}(\mathbf{e}_1); \dots; \text{vjp_f, x}(\mathbf{e}_n)]$. In our small example, $\mathbf{e}_1 = (1, 0)$ returns $(2, 0)$ and $\mathbf{e}_2 = (0, 1)$ returns $(2, 1)$: exactly the two rows. Why is this reformulation worth having? The book answers by looking at the parts of a fully-connected layer, linear projections and element-wise non-linearities, beginning with a linear projection without bias.

Remember

Reverse mode costs only vector-matrix products, linear in every size, but it must store every intermediate value of the forward pass. Gradient checkpointing keeps only some values and recomputes the rest during the backward pass; in the book's four-block example that costs 1.25 times the computation. The backward step needs only vector-Jacobian products, $\text{vjp}_f(\mathbf{v}) = \mathbf{v}^T \partial f(\mathbf{x})$, one for the input and one for the weights. So each primitive needs just its two VJPs, and no Jacobian is ever built in memory; the full Jacobian can still be recovered row by row with basis vectors.

Questions

1. What does gradient checkpointing trade against what?
 - a. It spends extra computation in the backward pass to store fewer intermediate values
 - b. It spends extra memory to avoid any recomputation
 - c. It replaces reverse mode by forward mode for the middle blocks
 - d. It drops some gradients to save time
2. In the book's figure, why does the backward pass cost 1.25 times the usual computation?
 - a. Because all four blocks are run forward twice

- b. Because the third block was not stored and has to be recomputed once from the second block's checkpoint
 - c. Because the backward pass is always a quarter more expensive than the forward pass
 - d. Because the fourth block's values were lost
3. What does a vector-Jacobian product give you, compared with the Jacobian itself?
- a. The full Jacobian, computed faster
 - b. Only the diagonal of the Jacobian
 - c. The product $\mathbf{v}^T \partial f(\mathbf{x})$, a vector the size of the input, without ever building the Jacobian
 - d. A vector the size of the output
4. Is anything lost by giving each primitive only its two VJPs instead of its Jacobians?
- a. Yes, the Jacobians can no longer be obtained at all
 - b. No, calling the VJP with the basis vectors $\mathbf{e}_1, \dots, \mathbf{e}_n$ returns the Jacobian one row at a time
 - c. Yes, the weight gradients become approximate
 - d. No, because VJPs and Jacobians always have the same size

Answers

1. **a)** Only a few outputs, the checkpoints, are kept; the others are recomputed from the nearest checkpoint during the backward pass, which costs more computation but less memory.
2. **b)** Four blocks forward plus one block recomputed makes five block computations instead of four, that is 1.25 times as much.

3. **c)** You hand the function a vector the size of its output and get back a vector the size of its input, as if the Jacobian had been used.
4. **b)** Each basis vector picks out one row, so stacking $\text{vjp_f}, \mathbf{x}(\mathbf{e}_1)$ to $\text{vjp_f}, \mathbf{x}(\mathbf{e}_n)$ rebuilds the input Jacobian.

Worked examples

The cost of the book's checkpointing example A model has four blocks. The forward pass keeps only the outputs of the first, second and fourth blocks. How much computation does the backward pass need compared with the standard case, counting block computations?

The forward pass computes the four blocks once: 4 block computations.

Going backwards, the pass reaches the third block and finds its values missing.

It reruns forward from the nearest checkpoint, the second block's output, through the third block: 1 more block computation.

Total 5 instead of 4, and $5 / 4 = 1.25$.

Answer: 1.25 times the computation.

A vector-Jacobian product by hand f takes $\mathbf{x} = (3, 4)$ to $(x_1 + x_2, x_1 x_2)$. Compute $\text{vjp_f}(\mathbf{v})$ for $\mathbf{v} = (2, 1)$.

The Jacobian has one row per output: the first output gives $(1, 1)$, the second gives $(x_2, x_1) = (4, 3)$.

$\mathbf{v}^T$ times the Jacobian is 2 times the first row plus 1 times the second.

$2 \times (1, 1) = (2, 2)$, and adding $(4, 3)$ gives $(6, 5)$.

Answer: $(6, 5)$, a vector of the input's size.

Practice

1. A model has 8 blocks. Only the outputs of blocks 3 and 6 are not stored, and each must be recomputed once during the backward pass.

How much computation does the backward pass need compared with the standard case?

Hint: Count the forward computations and add the recomputed blocks.

Answer: 8 forward computations plus 2 recomputed blocks is 10 instead of 8, that is 1.25 times as much.

2. f takes $\mathbf{x}$ to $(2x_1 + x_2, x_1 - x_2)$. Call its VJP with $\mathbf{e}_1 = (1, 0)$ and $\mathbf{e}_2 = (0, 1)$ and write the Jacobian you recover.

Hint: Each basis vector picks out one row of the Jacobian.

Answer: $\text{vjp}(\mathbf{e}_1) = (2, 1)$ and $\text{vjp}(\mathbf{e}_2) = (1, -1)$, so the Jacobian has rows $(2, 1)$ and $(1, -1)$.

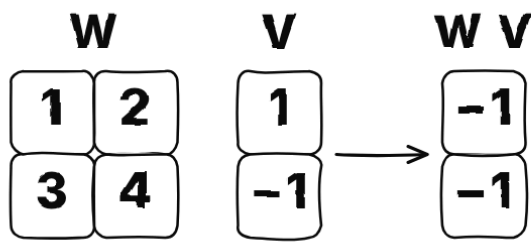
3. Explain in your own words why reverse mode needs much more memory than forward mode, even though it computes far less.

Hint: What does the backward pass need from the forward pass, and when?

Answer: Forward mode does its updates alongside the program and can forget each value once it has moved on. Reverse mode can only start once the whole program has run, and every backward step needs the values of its step, so all intermediate outputs must be kept until the backward pass reaches them.

The Backward Step of a Layer

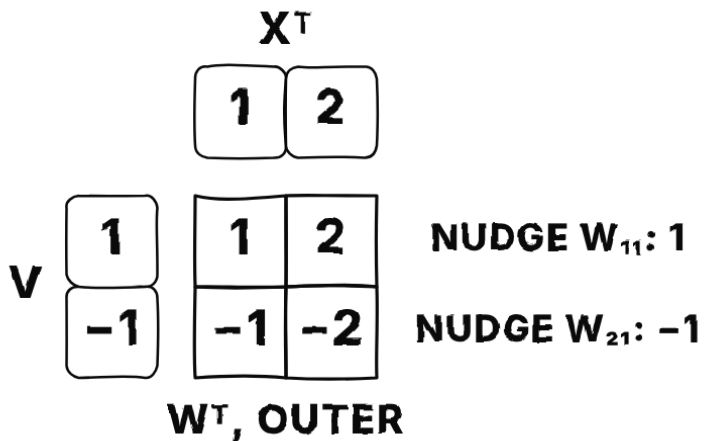
Set up a small experiment in a spreadsheet. Put a 2 by 2 weight table W with rows (1, 2) and (3, 4), an input x equal to (1, 2), and a vector v equal to (1, -1) that weights the two outputs. Compute W times x , which is (5, 11), and then the single number v times that, 1 times 5 minus 1 times 11, which is -6. Now nudge the top-left weight from 1 to 1.01 and watch the result: it moves by 0.01, from -6 to -5.99. Nudge the bottom-left weight instead and it moves by -0.01; there is a pattern here, and the book turns it into a formula.



Take the simplest part of a fully-connected layer, a linear projection without bias: $f(\mathbf{x}, \mathbf{W}) = \mathbf{W}\mathbf{x}$. Its input Jacobian is simply the matrix $\mathbf{W}$, but its weight Jacobian is a rank-3 tensor: one direction for the outputs and two for the rows and columns of $\mathbf{W}$. That is exactly the kind of array the book wants to avoid building. The VJPs avoid it.

The input VJP has no special structure: $\text{vjp}_{f,\mathbf{x}}(\mathbf{v}) = \mathbf{v}^T \mathbf{W}^T = [\mathbf{W}\mathbf{v}]^T$. In words: the adjoint coming back is multiplied by the transpose of the weights. The book writes this product with the weights transposed, and the result is again a vector. In our spreadsheet, $\mathbf{W}$ times $(1, -1)$ is $(1 - 2, 3 - 4)$, which is $(-1, -1)$, the vector the book's formula sends back towards the input.

The weight VJP turns out to be an outer product, with no rank-3 tensor anywhere: $\text{vjp}_{f,\mathbf{W}}(\mathbf{v}) = \mathbf{v}\mathbf{x}^T$. An outer product of two vectors is the table whose entry in row i and column j is the i -th entry of the first times the j -th entry of the second. The book works it out in a short box: the number $y = \mathbf{v}^T \mathbf{W} \mathbf{x} = \sum_i \sum_j W_{ij} v_i x_j$ contains each weight exactly once, multiplied by v_i and x_j . So $\partial y / \partial W_{ij} = v_i x_j$, which is the definition of the outer product, entry by entry. With our numbers, $\mathbf{v}$ times $\mathbf{x}$ transposed has rows $(1, 2)$ and $(-1, -2)$: the top-left entry, 1, is the 0.01 we saw for the top-left nudge, divided by 0.01, and the bottom-left, -1, matches the second nudge.



The Weight VJP Is an Outer Product

The rule to remember is short. Every time the forward pass applies a linear projection, the backward pass multiplies the travelling gradient by the transpose of its weights, and computes the gradient for $\mathbf{W}$ as an outer product. When a chat assistant is trained, most of its layers are such projections, and these two operations are much of what its backward pass does.

Now an element-wise activation function with no trainable parameters, such as the ReLU: $f(\mathbf{x}, \emptyset) = \varphi(\mathbf{x})$. With no weights there is only an input VJP. Its Jacobian is a diagonal matrix, zero everywhere except on the diagonal, where it holds the derivatives of φ : $[\partial_{\mathbf{x}} \varphi(\mathbf{x})]_{ii} = \varphi'(x_i)$, because each output depends only on its own input. A diagonal matrix times a vector is just a scaling, entry by

entry, which is called the Hadamard product: $\text{vjp_x}(f, \mathbf{v}) = \mathbf{v} \odot \phi'(\mathbf{x})$. For the ReLU, with $\mathbf{x}$ equal to $(2, -1, 3)$, the derivatives are $(1, 0, 1)$; a travelling vector $(0.5, 0.7, -0.2)$ comes out as $(0.5, 0, -0.2)$. Here too the full diagonal matrix is never built.

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

ReLU'

1
0
1

$\odot$

V

0.5
0.7
-0.2

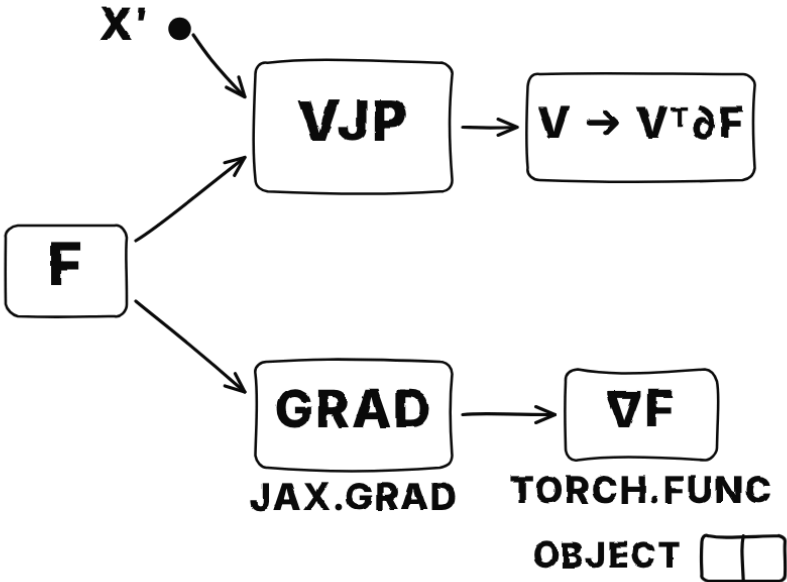
$=$

0.5
0
-0.2

ReLU's Backward Step Is a Mask

How do real frameworks organise all this? There are many ways, the book says, from Wengert lists, a recorded list of the operations performed, as in TensorFlow, to source-to-source transformations that rewrite the program's code itself. Writing primitives as functions of input and weights fits functional frameworks such as JAX, where everything is a function. There a VJP is a higher-order function, a function that takes and returns functions, with the signature $(\mathbb{R}^c \rightarrow \mathbb{R}^{c'}) \rightarrow \mathbb{R}^c \rightarrow (\mathbb{R}^{c'} \rightarrow \mathbb{R}^c)$. In words: give it a function f and an input

point, and it gives you back a new function, which takes a vector $\mathbf{v}$ of the output's size and returns $\mathbf{v}^T \partial f(\mathbf{x}')$.



A VJP Returns a Function

The gradient of a function with one output is another higher-order function, with signature $(\mathbb{R}^c \rightarrow \mathbb{R}) \rightarrow (\mathbb{R}^c \rightarrow \mathbb{R}^c)$: it takes f and returns a function that computes $\nabla f(\mathbf{x})$. JAX offers such functions, the book names `jax.grad` among them, and PyTorch copies the interface in its `torch.func` module. The book's example takes the sum of the squares of a vector, asks for its gradient function, applies it to ten random numbers, and gets back ten numbers, as it should: one per input, here twice each input. Such a function can also be traced, for instance by `torch.compile`, into an optimized computational graph, and composing VJPs and JVPs gives more, such as second derivatives. In

practice, though, models are built as objects that keep their parameters inside, and the book turns next to bridging the two styles.

Remember

For a linear projection $\mathbf{W}\mathbf{x}$, the weight Jacobian would be a rank-3 tensor, but the VJPs avoid it: the input VJP multiplies by the transpose of $\mathbf{W}$, the weight VJP is the outer product $\mathbf{v}\mathbf{x}^T$. The box's reasoning is simple: in $\mathbf{v}^T\mathbf{W}\mathbf{x}$ each weight W_{ij} appears once, multiplied by v_i and x_j . For an element-wise activation the Jacobian is diagonal, so its VJP is just $\mathbf{v} \odot \phi'(\mathbf{x})$, a scaling, again without building the matrix. In functional frameworks like JAX a VJP and a gradient are higher-order functions that take a function and return a new one; PyTorch offers the same in `torch.func`.

Questions

1. Why is the weight VJP of a linear projection $\mathbf{W}\mathbf{x}$ so much easier to handle than its weight Jacobian?
 - a. Because the weight Jacobian is a rank-3 tensor, while the weight VJP is just the outer product $\mathbf{v}\mathbf{x}^T$
 - b. Because the weight VJP ignores the input $\mathbf{x}$
 - c. Because the weight Jacobian is simply $\mathbf{W}$
 - d. Because the weight VJP is always a diagonal matrix

2. Why does the derivative of $y = \mathbf{v}^T \mathbf{W} \mathbf{x}$ with respect to W_{ij} come out as $v_i x_j$?
 - a. Because $\mathbf{W}$ is a square matrix
 - b. Because in the double sum each weight W_{ij} appears exactly once, multiplied by v_i and x_j
 - c. Because the input Jacobian is the transpose of $\mathbf{W}$
 - d. Because the ReLU sets the other entries to zero
3. For an element-wise activation such as the ReLU, what does the input VJP do to the travelling vector $\mathbf{v}$?
 - a. It multiplies $\mathbf{v}$ by the transpose of a weight matrix
 - b. It takes the outer product of $\mathbf{v}$ with the input
 - c. It scales each entry of $\mathbf{v}$ by the derivative of ϕ at the matching input, the Hadamard product $\mathbf{v} \odot \phi'(\mathbf{x})$
 - d. It leaves $\mathbf{v}$ unchanged, because there are no weights
4. In a functional framework like JAX, what does the VJP of a function return?
 - a. A number: the derivative at one point
 - b. A new function that takes a vector of the output's size and returns $\mathbf{v}^T \partial f(\mathbf{x}')$
 - c. The full Jacobian matrix of the function
 - d. A list of the operations the program performed

Answers

1. a) The weight Jacobian has one direction for the outputs and two for the rows and columns of $\mathbf{W}$; the weight VJP collapses it into a table with entries $v_i x_j$, the shape of $\mathbf{W}$ itself.

2. b) Written out, y is the sum of $W_{ij} v_i x_j$ over all i and j , so nudging one weight changes y by $v_i x_j$ times the nudge.

3. c) The Jacobian is diagonal with entries $\phi'(x_i)$, and a diagonal matrix times a vector is an entry-by-entry scaling, so the matrix never has to be built.

4. b) The VJP is a higher-order function: given f and an input point, it gives back a function that turns an output-sized vector into $\mathbf{v}^T \partial f(\mathbf{x}')$.

Worked examples

The weight gradient as an outer product Following the book's box, take $y = \mathbf{v}^T \mathbf{W} \mathbf{x}$ with $\mathbf{v} = (1, -1)$ and $\mathbf{x} = (1, 2)$. Write the gradient of y with respect to the 2 by 2 matrix $\mathbf{W}$.

Write y as the double sum of $W_{ij} v_i x_j$: each weight appears once.

So $\partial y / \partial W_{ij} = v_i x_j$, which is the outer product $\mathbf{v} \mathbf{x}^T$.

First row, $v_1 = 1$: $(1 \times 1, 1 \times 2) = (1, 2)$.

Second row, $v_2 = -1$: $(-1 \times 1, -1 \times 2) = (-1, -2)$.

Answer: A 2 by 2 table with rows $(1, 2)$ and $(-1, -2)$.

Practice

1. Compute the weight VJP $\mathbf{v} \mathbf{x}^T$ for $\mathbf{v} = (2, 1)$ and $\mathbf{x} = (3, 0, 1)$.

Hint: Row i is v_i times the whole of $\mathbf{x}$.

Answer: Rows $(6, 0, 2)$ and $(3, 0, 1)$: the first is 2 times $\mathbf{x}$, the second 1 times $\mathbf{x}$.

2. A ReLU receives $\mathbf{x} = (-2, 0.5, 4)$, and the backward pass brings the vector $\mathbf{v} = (3, 3, 3)$. What does its input VJP send back?

Hint: The ReLU's derivative is 0 for negative inputs and 1 for positive ones; scale entry by entry.

Answer: $(0, 3, 3)$: the derivatives are $(0, 1, 1)$, and $\mathbf{v} \odot \phi'(\mathbf{x})$ multiplies entry by entry.

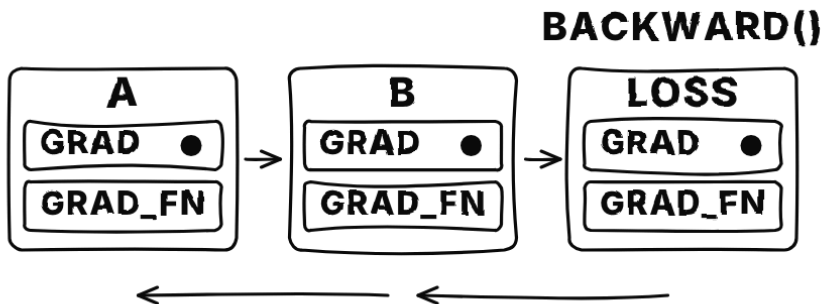
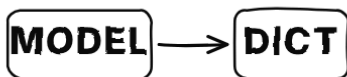
3. Explain why the Jacobian of an element-wise activation function is diagonal.

Hint: Ask which inputs each output depends on.

Answer: Each output $\phi(x_i)$ depends only on its own input x_i , so nudging any other input does not move it: all entries off the diagonal are zero, and the diagonal holds $\phi'(x_i)$.

Frameworks, and Why the ReLU Wins

Before the frameworks, one small bridge. In PyTorch a model is an object that keeps its parameters inside, while the functional style wants a function of input and weights. The book shows how to "purify" the object: collect the model's named parameters into a dictionary, then call the model's forward computation through `torch.func.functional_call`, passing the model, that dictionary and the input explicitly. Now the parameters are an argument again, as in the book's notation.

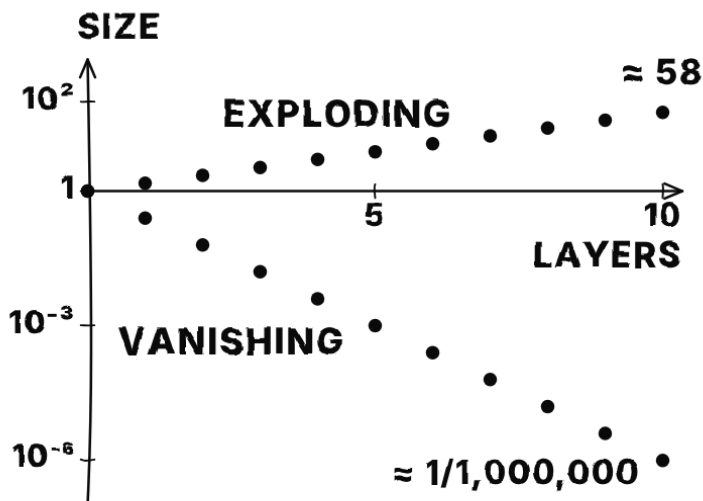


A Tensor Remembers Its Maker

More generally, PyTorch handles the object style directly, without such intermediate steps. The book's figure draws a tensor, PyTorch's array, as a box with extra fields attached. On the left, at creation, the tensor carries a slot for its gradient, empty at first, and a note of the operation that produced it, the `grad_fn` property. On the right, during a backward pass, these notes are followed in reverse through the computational graph, and each gradient is stored inside its tensor's `grad` field, but only where `requires_grad` has been explicitly set to true, so as not to waste memory. All this starts when you call `backward()` on a single number, the loss; it is the moment, in any photo app's training script, when the gradients for every weight appear. The author stresses that this is only an overview, points to the official documentation, and warmly suggests building a small reverse-mode system yourself, for instance by studying a teaching implementation called `micrograd`.

Try it yourself

Take a calculator, type 1, and multiply by 0.25 ten times. You might expect something small but usable. What you see is 0.00000095, about one in a million. Now start again from 1 and multiply by 1.5 ten times instead: you reach about 58, and fifty times would give over six hundred million.



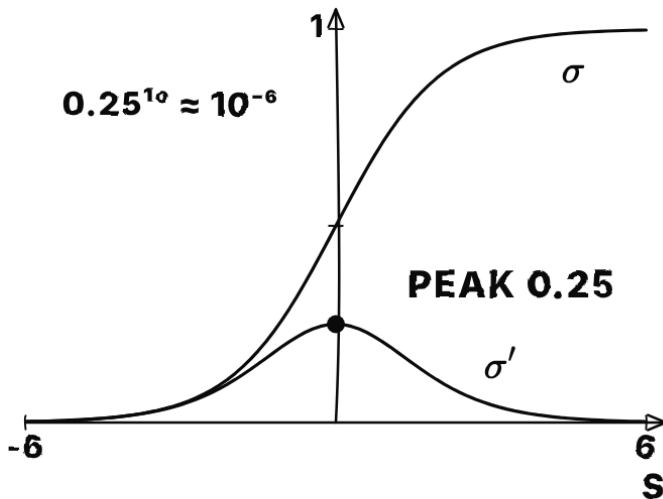
Ten Layers of 0.25 and of 1.5

That calculator game is the backward pass of a deep network. From the ReLU part we know that every activation function scales the adjoints by $\phi'(\mathbf{x})$ on their way back. If $\phi'(\cdot) < 1$ everywhere, the gradient shrinks towards zero exponentially fast in the number of layers: the vanishing gradient problem. If $\phi'(\cdot) > 1$ everywhere, it grows exponentially towards infinity: the exploding gradient problem. These are serious in practice, because libraries store numbers with limited precision, typically 32 bits or fewer, so a few dozen layers are enough for a number to underflow to zero or overflow.

The book adds a curious box. A stack of purely linear layers, computed in floating point, is not exactly linear, because of tiny jumps at the limits of machine precision. This is generally no problem, but

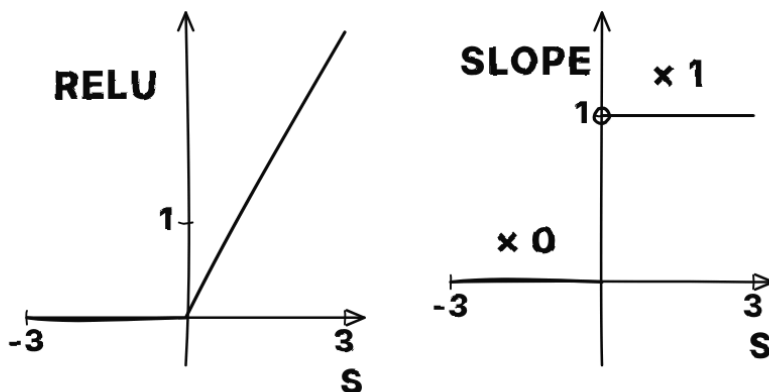
the author notes it has been exploited to train deep networks made only of linear layers.

Now the sigmoid, the S-shaped curve $\sigma(s)$ that was popular in the past as a soft version of the step function. Its derivative is $\sigma'(s) = \sigma(s)(1 - \sigma(s))$. In words: the slope is the value times what is left to reach 1. Since σ always lies between 0 and 1, this product is largest when σ is one half, which gives one half times one half: so $\sigma'(s) \in [0, 0.25]$. At s equal to 0, σ is 0.5 and the slope is exactly 0.25; that is our calculator's factor, the best case, and ten sigmoid layers already shrink the gradient a millionfold. The book's plot shows it: the red sigmoid climbs from 0 to 1, while its green derivative is a low hump peaking at a quarter and fading to nothing on both sides.



The Sigmoid's Slope Peaks at a Quarter

Could an activation function avoid both problems? Only the identity, which leaves everything unchanged, has $\phi'(s) = 1$ everywhere, and the identity would make the layers collapse into one linear map. So we need a function linear enough to keep gradients healthy, yet non-linear enough to separate the layers. The ReLU fits: $\partial_s \text{ReLU}(s) = 0$ for $s < 0$, 1 for $s > 0$. Either the gradient is zeroed, which makes the computation sparse, or it is multiplied by exactly 1, with no scaling at all; in the book's second plot the red ReLU is flat and then rises along a diagonal, and the green derivative is a step from 0 to 1.



OVERWRITE INPUT

The ReLU's Slope Is 0 or 1

One more gift of the ReLU. Its gradient is the same whether you compute it from the layer's input or from its output, because the ReLU only masks the negative values and leaves the positive ones

untouched: a positive output means a positive input. So in reverse mode the layer's input can be overwritten by its output during the forward pass, saving a little memory without harming the correctness of automatic differentiation; PyTorch offers this through a parameter for working in place.

Remember

In PyTorch each tensor remembers the operation that made it; calling backward on the loss follows those notes in reverse and stores gradients inside the tensors that ask for them. Every activation function multiplies the backward gradient by its derivative, so factors below 1 make gradients vanish and factors above 1 make them explode, exponentially in the number of layers, and limited precision makes this bite quickly. The sigmoid's derivative $\sigma(s)(1 - \sigma(s))$ is at most 0.25, a prime cause of vanishing gradients. The ReLU's derivative is 0 or exactly 1, so it neither shrinks nor inflates what passes, and it even lets the forward pass overwrite its input to save memory.

Questions

1. In PyTorch, what lets a call to `backward()` find its way back through the computation?
 - a. Each tensor carries a note of the operation that created it, and these notes are followed in reverse
 - b. PyTorch reruns the whole program symbolically

- c. The gradients are computed numerically, two calls per weight
 - d. Every tensor stores the full Jacobian of the step that made it
2. A deep network uses an activation whose derivative is below 1 everywhere. What happens to the gradients reaching the first layers?
- a. They grow exponentially with the number of layers
 - b. They stay the same, because the weights compensate
 - c. They shrink towards zero exponentially fast with the number of layers
 - d. They become exactly 1
3. Why is the sigmoid a prime candidate for vanishing gradients?
- a. Because its derivative can be larger than 1
 - b. Because its derivative $\sigma(s)(1 - \sigma(s))$ never exceeds 0.25
 - c. Because it is not differentiable at 0
 - d. Because it is a linear function
4. Why may the ReLU overwrite its input with its output in the forward pass without breaking reverse mode?
- a. Because the ReLU has no derivative to compute
 - b. Because its gradient can be read just as well from the output: positive outputs come from positive inputs, and negatives are masked
 - c. Because PyTorch stores a second copy anyway
 - d. Because the backward pass never visits activation layers

Answers

1. **a)** Tensors are augmented with the operation that generated them (`grad_fn`); `backward()` follows these in reverse and stores gradients inside the tensors that require them.
2. **c)** Each activation scales the adjoints by ϕ' on the way back, so factors below 1 multiply up into a vanishing gradient.
3. **b)** With σ between 0 and 1, the product $\sigma(1 - \sigma)$ is at most one half times one half, so every sigmoid layer shrinks the gradient at least fourfold.
4. **b)** The ReLU only masks negative values and leaves positive ones untouched, so its gradient is the same whether computed from input or output.

Worked examples

The sigmoid's largest slope Using $\sigma'(s) = \sigma(s)(1 - \sigma(s))$ and the fact that $\sigma(s)$ lies between 0 and 1, find the largest possible value of σ' and the factor by which ten sigmoid layers shrink a gradient at best.

Write the derivative as a product of two numbers that add up to 1: σ and $1 - \sigma$.

Such a product is largest when the two are equal, at $\sigma = 0.5$, giving $0.5 \times 0.5 = 0.25$.

So $\sigma'(s)$ lies between 0 and 0.25, and each sigmoid layer multiplies the backward gradient by at most 0.25.

Over ten layers: 0.25 to the tenth power, about 0.00000095.

Answer: At most 0.25 per layer; ten layers shrink the gradient to about one millionth at best.

Practice

1. At some input the sigmoid's value is $\sigma = 0.8$. What is its slope there?

Hint: Multiply the value by what is left to reach 1.

Answer: $0.8 \times 0.2 = 0.16$, below the maximum of 0.25.

2. Suppose every layer of a ten-layer network multiplied the backward gradient by 2. By what factor would the gradient at the first layer be scaled? What is this problem called?

Hint: Raise the factor to the number of layers.

Answer: 2 to the tenth power, 1,024: the gradient explodes, which is the exploding gradient problem.

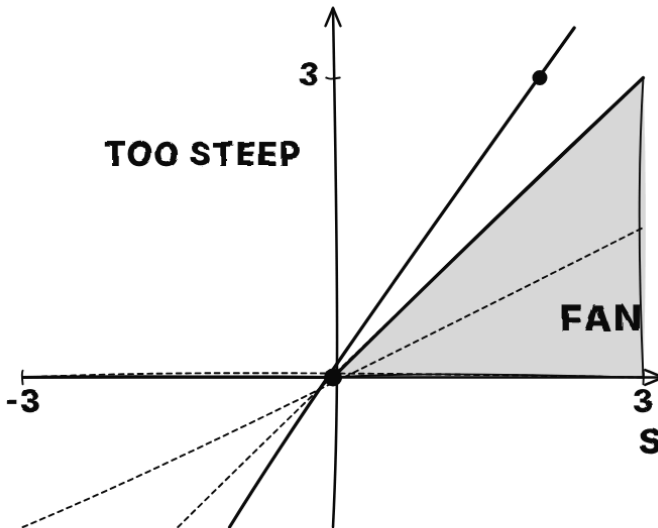
3. Explain why the ReLU avoids both vanishing and exploding gradients for the inputs where it is active.

Hint: Look at the two values its derivative can take.

Answer: Its derivative is 0 for negative inputs and exactly 1 for positive ones. Where it is active, the gradient is multiplied by exactly 1, so it is neither shrunk nor inflated; elsewhere it is set to zero, which only makes the computation sparse.

The Corner of the ReLU

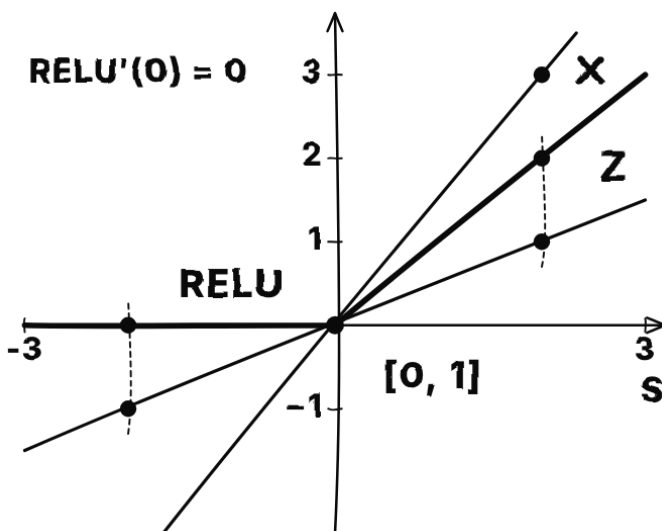
Draw the ReLU on a sheet of paper: flat along zero for negative numbers, then rising along the diagonal for positive ones, with a sharp corner at the origin. Now lay a ruler through that corner. Tilt it flat, and the whole graph lies on or above it; tilt it up to the diagonal, and the graph still lies on or above it. Any tilt in between works too, but tilt it steeper than the diagonal and the graph dips below it on the right. At the corner there is no single slope, but a whole fan of them, and that fan is how the book settles a question it has left open until now.



A Fan of Rulers at the Corner

The ReLU has no derivative at 0, so a network built with it is not smooth. What then? The book first gives the pragmatic answer: training with stochastic gradient descent from a random, non-zero starting point, the chance of landing exactly on 0 is practically nil, while the gradient exists at $\text{ReLU}(\epsilon)$ for any ϵ other than 0, however small. In a chat assistant's training run, with billions of numbers flowing through, an exact zero is a rare accident.

The technical answer is the subgradient. For a convex function f , one whose graph bends upwards everywhere, a subgradient at x is a number z such that, for every y , $f(y) \geq f(x) + z(y - x)$. In words: the straight line through the point $(x, f(x))$ with slope z stays on or below the whole graph, like our ruler. It echoes the definition of convexity, and where f has a derivative only one such line exists, the tangent, so the subgradient is just the derivative. At a corner many slopes qualify, and together they form the subdifferential: $\partial_x f(x) = \{z \mid z \text{ is a subgradient of } f(x)\}$.

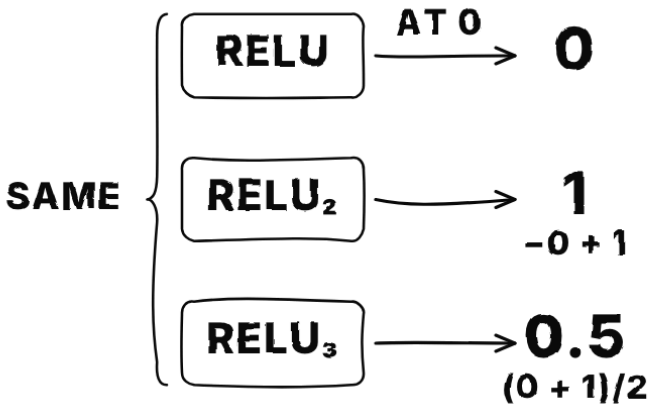


Checking Slopes 0.5 and 1.5

Now the ReLU can be completed by replacing its gradient with its subdifferential at 0: $\partial_s \text{ReLU}(s) = \{0\}$ for $s < 0$, $\{1\}$ for $s > 0$, $[0, 1]$ for $s = 0$. Check one value with numbers: take z equal to 0.5 at the corner. At y equal to 2 the line gives 1 and the ReLU gives 2; at y equal to -2 the line gives -1 and the ReLU gives 0; the line stays below, as it should. Take z equal to 1.5 and at y equal to 2 the line gives 3, above the ReLU's 2: not a subgradient, just as the steep ruler showed. So any value from 0 to 1 is valid at the corner, and most implementations pick $\text{ReLU}'(0) = 0$; choosing a subgradient at every step of descent is called subgradient descent.

The full story is trickier. For functions that are not convex, a subgradient need not exist, and one turns to generalizations that relax

the definition to a small neighbourhood of x , such as the Clarke subdifferential. Automatic differentiation adds its own trap: different implementations of the same function can return different subgradients, some not even valid, and a formal proof needs more refined chain rules. The book's note gives an example. Define $\text{ReLU}_2(s) = \text{ReLU}(-s) + s$ and $\text{ReLU}_3(s) = 0.5(\text{ReLU}(s) + \text{ReLU}_2(s))$. Try a few numbers and both give exactly the ReLU, yet a backward pass in PyTorch at 0 returns 0 for the ReLU, 1 for ReLU₂ and 0.5 for ReLU₃. The reason is the chain rule at work: ReLU₂'s derivative is minus ReLU'(0), which is 0, plus 1, and ReLU₃ averages 0 and 1.



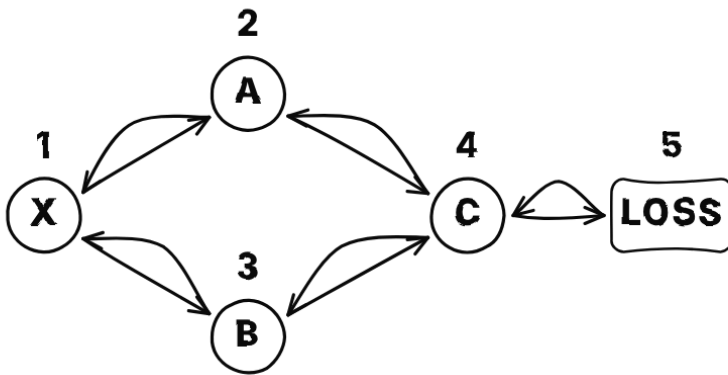
Same Function, Three Answers

From here the book turns to practice. If you did the exercises of the previous chapter, you have already used reverse mode in PyTorch and

JAX, and this chapter explains what happened inside. The author suggests rebuilding a small reverse-mode system yourself, for scalar values, with micrograd as a fine teaching model. One detail the chapter skipped: on a general graph without loops, the variables must be put in order before the backward pass, or gradients flow along wrong paths; micrograd does this with a cheap topological sort, an ordering in which every value comes after everything it depends on. Another exercise is to write a new primitive in PyTorch by giving its forward pass and its VJPs, for instance a trainable activation function from the previous chapter, though PyTorch could also work out that backward pass by itself. In JAX the same paths exist: a teaching version of JAX called autodidax, writing a primitive with its own VJP, and the JAX Autodiff Cookbook for higher derivatives, Hessians and more.

MICROGRAD

AUTODIDAX



Order the Graph, Then Walk Back

That closes the chapter: from a spreadsheet chain to the engine that trains every model on your phone, all built on one idea, multiplying backwards.

Remember

The ReLU has no derivative at 0, but training almost never lands exactly there, and the gradient exists everywhere else. A subgradient at x is a slope z whose line through $(x, f(x))$ stays below the whole graph; at a smooth point it is the derivative, at a corner there is a whole set, the subdifferential. For the ReLU at 0 every value from 0 to 1 is valid, implementations usually take 0, and different but equivalent formulas can give different answers in a framework, as ReLU2 and ReLU3 show. The book ends by inviting you to build reverse mode yourself, with micrograd or autodidax, and to write primitives with their own VJPs.

Questions

1. What is the book's pragmatic answer to the ReLU having no derivative at 0?
 - a. Training starts from a random non-zero point, so landing exactly on 0 is practically impossible, and the gradient exists everywhere else
 - b. The ReLU is replaced by the sigmoid near 0
 - c. The network is smoothed before training
 - d. Frameworks refuse to differentiate at 0 and stop training
2. At a point where a convex function has a derivative, how many subgradients does it have?
 - a. None

- b. Exactly one, the derivative itself
 - c. A whole interval of them
 - d. Two, one from each side
3. Why is a slope of 1.5 not a subgradient of the ReLU at 0?
- a. Because subgradients must be negative
 - b. Because the line with slope 1.5 through the corner rises above the ReLU for positive inputs, for instance 3 against 2 at $y = 2$
 - c. Because only 0 is a valid subgradient at the corner
 - d. Because the ReLU is not convex
4. $\text{ReLU}_2(s) = \text{ReLU}(-s) + s$ equals the ReLU everywhere. Why does PyTorch's backward pass at 0 return 1 for it, but 0 for the ReLU?
- a. Because ReLU_2 is a different function from the ReLU
 - b. Because PyTorch uses numerical differentiation for ReLU_2
 - c. Because the chain rule combines the implementation's choice $\text{ReLU}'(0) = 0$ with the derivative of the added s , giving $0 + 1$
 - d. Because ReLU_2 is not continuous at 0

Answers

- 1. a)** With stochastic gradient descent from a random non-zero initialization, the probability of hitting $s = 0$ exactly is practically null, while the gradient is defined at any nonzero input.
- 2. b)** Where f is differentiable only one line through the point stays below the whole graph, the tangent, so the only subgradient is the derivative.
- 3. b)** A subgradient's line must stay on or below the whole graph; at $y = 2$ the steep line gives 3 while the ReLU gives 2.

4. c) The backward pass follows the formula as written: minus $\text{ReLU}'(0)$, which is taken as 0, plus 1 from the added s , so equivalent formulas can return different subgradients.

Worked examples

The book's ReLU₂ and ReLU₃ at zero With the implementation's choice $\text{ReLU}'(0) = 0$, follow the chain rule to find what a backward pass returns at $s = 0$ for $\text{ReLU}_2(s) = \text{ReLU}(-s) + s$ and $\text{ReLU}_3(s) = 0.5(\text{ReLU}(s) + \text{ReLU}_2(s))$.

For ReLU_2 , the first term $\text{ReLU}(-s)$ gives minus $\text{ReLU}'(0)$, that is 0, because of the inner minus sign.

The second term, s , gives 1, so ReLU_2 returns $0 + 1 = 1$.

For ReLU_3 , the backward pass averages the two pieces: 0.5 times $(0 + 1)$.

That gives 0.5, while the plain ReLU gives 0.

Answer: 1 for ReLU_2 and 0.5 for ReLU_3 , against 0 for the ReLU.

Practice

1. Check whether $z = 0.3$ is a subgradient of the ReLU at 0, by testing $y = 4$ and $y = -4$ in $f(y) \geq f(0) + z(y - 0)$.

Hint: Compute the line's value $0.3y$ and compare with $\text{ReLU}(y)$.

Answer: At $y = 4$ the line gives 1.2 and the ReLU 4; at $y = -4$ the line gives -1.2 and the ReLU 0. The line stays below in both cases, and indeed every z between 0 and 1 is a subgradient at 0.

2. Show that $\text{ReLU}_2(s) = \text{ReLU}(-s) + s$ equals the ReLU, by trying $s = 3$ and $s = -3$.

Hint: Compute $\text{ReLU}(-s)$ first, then add s .

Answer: For $s = 3$: $\text{ReLU}(-3) = 0$, plus 3 gives $3 = \text{ReLU}(3)$. For $s = -3$: $\text{ReLU}(3) = 3$, plus -3 gives $0 = \text{ReLU}(-3)$.

3. Explain why a reverse-mode system on a general graph without loops must order its variables before the backward pass.

Hint: Think of a value that feeds two later steps.

Answer: A value's adjoint is complete only after every step that uses it has sent its contribution back. Processing the variables in a topological order, each after everything it depends on, and walking that order in reverse, makes sure no gradient is passed on before it is complete; otherwise gradients would follow wrong paths. micrograd does this with a cheap topological sort.