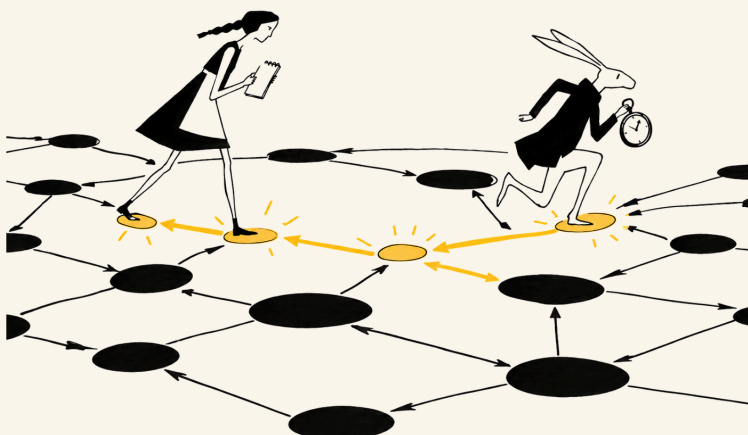


Alice's Adventures in a Differentiable Wonderland



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 Chain of Buttons

One Long Line of Numbers

Three Ways to Find a Slope

Walking the Chain Backwards

Never Build the Table

Notes Left Along the Way

A Quarter, Ten Times Over

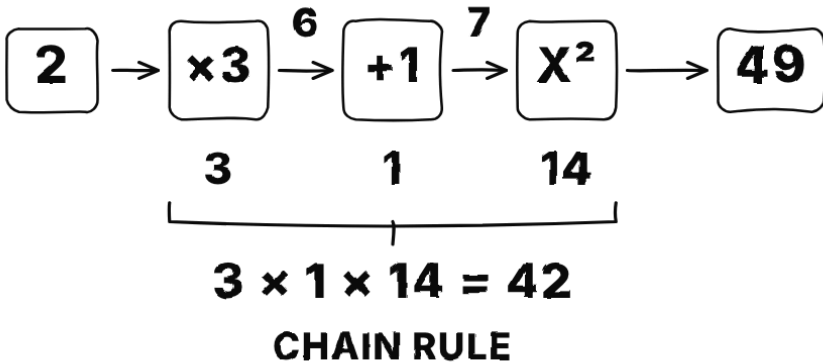
A Ruler at the Corner

A Chain of Buttons

Take a calculator and press a short chain of buttons: start with two, multiply by three, add one, and square the result. You get forty-nine. Now start again from two point zero one, a nudge of one hundredth, and press the same buttons. You might expect the answer to move by about a hundredth too, but it comes out near forty-nine point four two, so it moved forty-two times as much as your nudge. Where does forty-two come from?

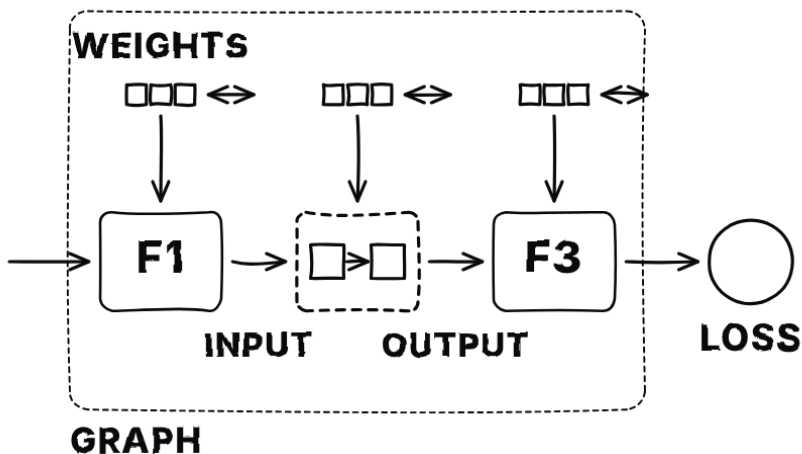
2.01

49.42



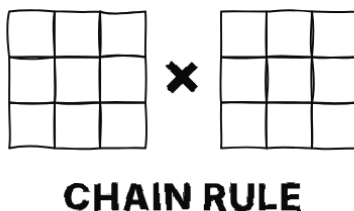
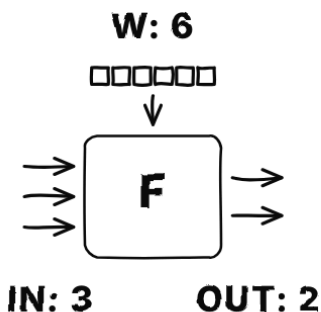
Look at each button on its own. Multiplying by three makes any small change three times bigger. Adding one passes a change through untouched, so its rate is one. Squaring, at the value seven, makes a small change fourteen times bigger, twice the value it works on. Multiply the three local rates, three times one times fourteen, and you get forty-two: the rate of the whole chain is the product of the rates of its links. That is the chain rule, and this chapter is about doing exactly this, quickly, for programs with millions of buttons.

When a chat assistant or the photo app on your phone is trained, the program is a long chain of such steps, and at the end stands a single number, the loss, which scores how badly the program did on a batch of examples. Training needs to know how that one score moves when each of the program's adjustable numbers, its weights, is nudged. Computing all those rates automatically from the program itself is called automatic differentiation. The book pictures the program as a computational graph: the set of small, indivisible operations the program actually performs when it runs, which it calls primitives. For simplicity the chapter treats the steps as a straight chain, one after another, but everything carries over to programs that branch and merge.



A Program as a Chain of Primitives

Each primitive takes in a list of numbers, the input, uses its own list of adjustable numbers, its weights, and hands on a new list, the output. A primitive can be as small as one multiplication, as large as a whole layer of the network, or even a block made of many layers; building big operations out of small ones is what programming always does. The book assumes only one thing about each primitive: that we know its local rates, collected in two tables called Jacobians. The input Jacobian says how each output number moves when each input number is nudged, so it has one row per output and one column per input. The weight Jacobian says the same for the weights, with one row per output and one column per weight.



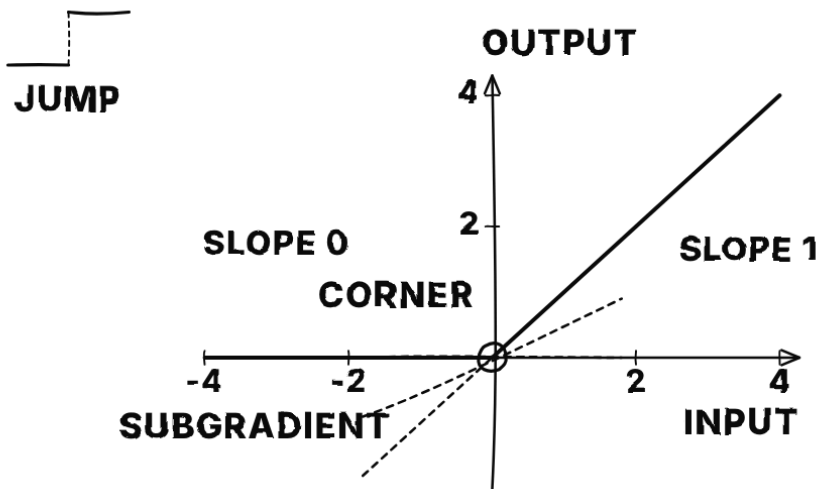
The Two Tables of a Primitive

Picture a primitive that takes three numbers in, holds six weights and gives two numbers out. Its input Jacobian is a table of two rows and three columns, six rates in all; its weight Jacobian has two rows and six columns, twelve rates. Now imagine an input of a thousand numbers and an output of a thousand: the input Jacobian alone holds a million rates, and that is one layer of a modest network.

At first the whole task looks trivial. The chain rule, as you saw with the calculator, says the rate of a chain is just the product of the tables of its links, multiplied one after the other. The hard part, and the subject of the chapter, is doing these multiplications efficiently. The answer the chapter arrives at is called reverse-mode automatic differentiation, or back-propagation in the language of neural

networks, and it is the cornerstone of how networks are trained. Understanding it also explains how the programs used to train networks, such as PyTorch, JAX or TensorFlow, are built, and why they differ.

One worry remains: what if a step has no rate at some point? The ReLU, a common step that keeps positive numbers and turns negative ones into zero, has a sharp corner at zero. Steps like this are handled with a weaker notion of slope, the subgradient, which comes at the end of the chapter. Steps that jump, like drawing a random sample or cutting at a threshold, can be brought in by replacing their rate with a smooth stand-in; the author leaves that to the next volume.



The ReLU and Its Corner

Remember

Automatic differentiation computes, straight from a program, how a single final score changes when each adjustable number of the program is nudged. The program is seen as a chain of primitives, each taking an input and its own weights and giving an output. For each primitive we only need its two tables of local rates: the input Jacobian, outputs against inputs, and the weight Jacobian, outputs against weights. The chain rule multiplies these tables, as three times one times fourteen gave forty-two on the calculator; the real challenge is doing it efficiently, and back-propagation is the answer.

Questions

1. On the calculator, the final answer moved forty-two times as much as the nudge. Why forty-two?
 - a. Because forty-two is the sum of the local rates three, one and fourteen, plus the start value
 - b. Because the local rates of the three buttons, three, one and fourteen, multiply to forty-two
 - c. Because squaring always multiplies a change by the final answer divided by the start
 - d. Because a nudge of one hundredth is always magnified by the number of buttons pressed

2. A primitive gives two numbers out and holds six weights. What shape does its weight Jacobian have?
 - a. Two rows and six columns, one row per output and one column per weight
 - b. Six rows and two columns, one row per weight and one column per output
 - c. Two rows and two columns, because only the outputs count
 - d. Six rows and six columns, because every weight affects every weight
3. If the chain rule just multiplies tables, why does the chapter call the task hard?
 - a. Because the chain rule only works for programs that branch and merge
 - b. Because the local rates of most primitives are unknown and must be guessed
 - c. Because the tables can be huge, so doing the multiplications efficiently is the real challenge
 - d. Because the final score is a list of numbers rather than a single number
4. Why does the ReLU need special treatment in this picture?
 - a. Because it has weights that cannot be nudged
 - b. Because it jumps at random, like drawing a sample
 - c. Because it changes the size of the list of numbers it receives
 - d. Because it has a sharp corner at zero, where it has no ordinary rate

Answers

1. **b)** Each button changes a small nudge by its own local rate, and the rate of the whole chain is the product of the rates of its links: three times one times fourteen.
2. **a)** The weight Jacobian says how each output moves when each weight is nudged, so it has one row per output and one column per weight: two by six, twelve rates.
3. **c)** A single layer with a thousand inputs and a thousand outputs already has a million rates in its input Jacobian, so the hard part is doing the multiplications efficiently, which back-propagation solves.
4. **d)** The ReLU keeps positive numbers and turns negative ones into zero, which leaves a sharp corner at zero; there a weaker notion of slope, the subgradient, is used.

Worked examples

Sizing the two tables A primitive takes four numbers in, holds ten weights and gives three numbers out. How many rates are in its input Jacobian, and how many in its weight Jacobian?

The input Jacobian has one row per output and one column per input, so it has three rows and four columns.

Three times four gives twelve rates.

The weight Jacobian has one row per output and one column per weight, so three rows and ten columns.

Three times ten gives thirty rates.

Answer: Twelve rates in the input Jacobian and thirty in the weight Jacobian.

A chain of three buttons Start from one, multiply by five, add two, and square. By how many times does the final answer move compared with a small nudge of the starting number?

Multiplying by five makes a small change five times bigger: local rate five.

Adding two passes the change through untouched: local rate one.

The value reaching the square is seven, and squaring makes a small change twice the value bigger: local rate fourteen.

The rate of the chain is the product of the local rates: five times one times fourteen.

Answer: Seventy times: a nudge of one hundredth moves the answer by about seven tenths, from forty-nine to about forty-nine point seven.

Practice

1. A layer takes a hundred numbers in, gives ten numbers out and holds one thousand and ten weights. How many rates are in its input Jacobian and in its weight Jacobian?

Hint: Rows are always outputs; columns are inputs for one table and weights for the other.

Answer: One thousand and ten thousand one hundred: ten rows by a hundred columns for the input Jacobian, ten rows by one thousand and ten columns for the weight Jacobian.

2. A chain has two buttons: multiply by two, then multiply by four. If you nudge the starting number by one tenth, how much does the final answer move?

Hint: Find each button's local rate first, then combine them the way the chain rule says.

Answer: By eight tenths: the local rates two and four multiply to eight, and eight times one tenth is eight tenths.

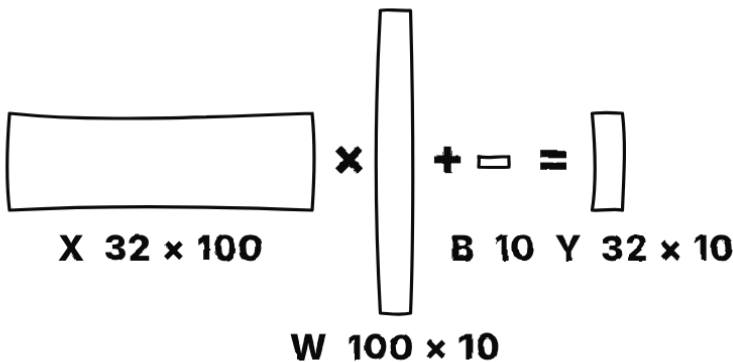
3. Explain in your own words why the chapter only needs to know two tables of local rates for each primitive, and not the rates of the whole program.

Hint: Think of what the chain rule does with the rates of the links.

Answer: The chain rule builds the rate of the whole chain by multiplying the rates of its links. So if every primitive's input Jacobian and weight Jacobian are known, the rates of the final score with respect to every weight follow; the work left is to do those multiplications efficiently.

One Long Line of Numbers

Take a sheet of squared paper and fill a block of it with numbers, thirty-two rows of a hundred numbers each. Now copy them onto one long strip, row after row, the first row, then the second right after it, and so on. Nothing has been lost: the strip holds the same three thousand two hundred numbers, and anyone who knows the rule can fold it back into the block. This small trick, unrolling a table into a single line, is what lets the chapter talk about every layer of a network with the same plain tables of rates.



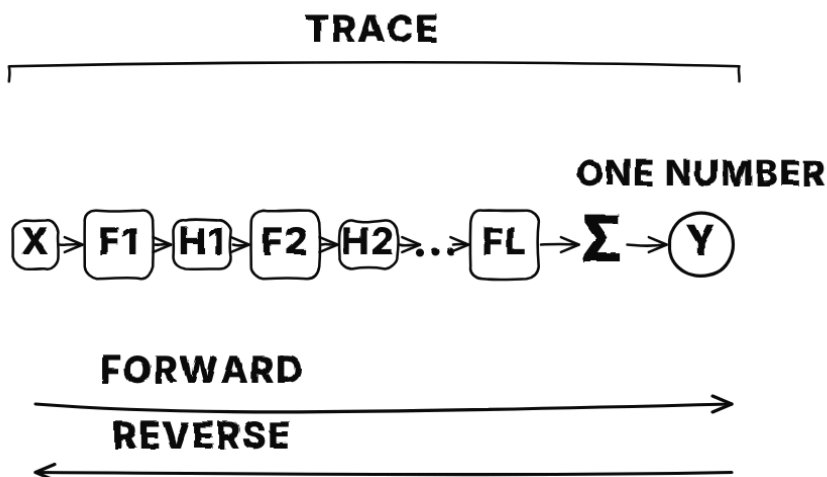
Here is why it is needed. When the photo app on your phone is trained, the pictures do not go through one at a time; they arrive in a mini-batch, say thirty-two at once. A basic fully-connected layer then works on a whole table: thirty-two rows, one per picture, each row holding a hundred features of that picture. The layer multiplies this table by a table of weights, a hundred rows by ten columns, and adds ten extra numbers called biases, one per output. Out comes a table of thirty-two rows and ten columns, ten new features for each picture.

Now ask the question from the calculator again: how does each output number move when each input number is nudged? An output is picked by two positions, its picture and its feature, and so is an input, so the table of rates has four directions instead of two. The book says these Jacobians have rank four, and they are hard to picture. The way out is the strip. Unroll the input block into one line of three thousand two hundred numbers, unroll the output block into one line of three hundred and twenty, and put the weights and the biases together into one line too. Then the input Jacobian is again a plain table, one row per output number and one column per input number: three hundred and twenty rows by three thousand two hundred columns, a little over a million rates.

Two things follow, and the book stresses both. First, from now on, when it speaks of the size of the input, it means the count of all its numbers together, the batch included, so here three thousand two hundred, not a hundred. Second, look at that million again, for one modest layer and one small batch. We may know exactly how to compute such a table and still not want to build it in memory, and this worry will shape everything that follows.

The book adds a note on how programs are written. Its way of describing a primitive, a function that receives its input and its weights side by side, is how a functional library such as JAX works. In PyTorch or TensorFlow a layer is an object that keeps its weights inside itself, and you call the object on the input alone. That style makes some things easier, such as waiting to create the weights until the first input shows how big they must be, which is called lazy initialization; but it adds a layer of wrapping to see through, and the two styles will end up computing rates in different ways.

Now the task itself. The program is a chain of steps written out one by one: the first primitive takes the input and its own weights and makes a first intermediate result; the second takes that result and its own weights and makes the next; and so on to the last. Such a written-out run is called an evaluation trace. The early steps are the layers of the model, the last step scores each example, for instance with the cross-entropy, and a final sum adds the scores of the batch into one number. So the program always ends in a single number, because training needs one number to push down.



An Evaluation Trace Ending in One Number

Automatic differentiation, in the book's definition, is computing, all at once and efficiently, how that one final number moves with respect to every weight of every step, knowing only the chain and the two local tables of each primitive. There are two great families of methods for this, forward mode and reverse mode, and they differ only in the order in which the local tables are multiplied together. The reverse one, back-propagation, will turn out to be far more efficient here. The same reasoning extends to programs that branch and merge, and to weights shared by several steps, which come back much later in the book.

Remember

Tables of numbers with several directions, such as a mini-batch of pictures, are unrolled into one long line so that every Jacobian stays a plain table of rows and columns. After unrolling, the size of an input means all of its numbers, batch included, and even one modest layer gives a table of about a million rates that we would rather not build. A program is written as a chain of primitives ending in a sum of per-example scores, so its result is always one number. Automatic differentiation computes how that number moves with every weight at once, and forward mode and reverse mode differ only in the order of the multiplications.

Questions

1. Why does the book unroll a mini-batch table into one long line of numbers?
 - a. So that every Jacobian becomes a plain table of rows and columns instead of a four-direction block
 - b. So that the layer needs fewer weights and runs faster on the phone
 - c. So that the pictures of the batch can be processed one at a time again
 - d. So that the biases can be dropped from the layer without changing its output

2. After unrolling, what does the book mean by the size of the input of a layer working on thirty-two pictures with a hundred features each?
 - a. A hundred, the number of features of one picture
 - b. Thirty-two, the number of pictures in the batch
 - c. Three thousand two hundred, all the numbers together, batch included
 - d. One hundred and thirty-two, the features plus the pictures
3. Why does the program in the problem statement always end in a single number?
 - a. Because every primitive in the chain gives exactly one number out
 - b. Because a final sum adds the scores of all examples of the batch, and training needs one number to push down
 - c. Because the evaluation trace only records the last layer of the model
 - d. Because the weights of all steps are shared and add up to one number
4. What separates forward mode from reverse mode?
 - a. Forward mode works for PyTorch, reverse mode only for JAX
 - b. Forward mode needs the input Jacobians, reverse mode only the weight Jacobians
 - c. Forward mode handles branching programs, reverse mode only straight chains
 - d. Only the order in which the local tables of rates are multiplied together

Answers

1. **a)** An output and an input are each picked by two positions, so the table of rates has four directions; unrolling both into lines turns it back into a plain table, one row per output number and one column per input number.
2. **c)** From now on the size of an input is the count of all its numbers together, the batch included: thirty-two times a hundred.
3. **b)** The last step scores each example and a final sum adds the scores of the batch into one number, which is what training pushes down.
4. **d)** Both compute the same rates from the same local tables; they differ only in the order of the multiplications, and the reverse order will prove far more efficient.

Worked examples

The tables of one layer A fully-connected layer receives a mini-batch of thirty-two pictures with a hundred features each and gives ten features per picture, using a hundred-by-ten table of weights. After unrolling, how big are its input Jacobian and its weight Jacobian, counting only the hundred-by-ten weights as the book's shapes do?

Unroll the input: thirty-two times a hundred gives three thousand two hundred numbers.

Unroll the output: thirty-two times ten gives three hundred and twenty numbers.

The input Jacobian has one row per output number and one column per input number: three hundred and twenty by three thousand two hundred, which is one million twenty-four thousand rates.

The weights unroll into a hundred times ten, a thousand numbers, so the weight Jacobian is three hundred and twenty by a thousand, three hundred and twenty thousand rates.

Answer: The input Jacobian holds one million twenty-four thousand rates and the weight Jacobian three hundred and twenty thousand.

Practice

1. A layer works on a mini-batch of eight examples with fifty features each and gives twenty features per example, using a fifty-by-twenty table of weights. After unrolling, how many rates are in its input Jacobian, and how many in its weight Jacobian, counting only those weights?

Hint: First count all the numbers of the input, of the output and of the weights, batch included.

Answer: Sixty-four thousand and one hundred and sixty thousand: the output has eight times twenty, a hundred and sixty numbers, the input eight times fifty, four hundred, and the weights a thousand; a hundred and sixty times four hundred is sixty-four thousand, a hundred and sixty times a thousand is a hundred and sixty thousand.

2. Explain in your own words why the book says we may know how to compute a Jacobian and still not want to build it in memory.

Hint: Think of how many rates even one modest layer with a small batch has.

Answer: Because after unrolling, even one modest layer on a small batch has a Jacobian of about a million rates. Computing it is possible, but storing such tables for every layer would take a great deal of memory, so a good method should avoid building them in full.

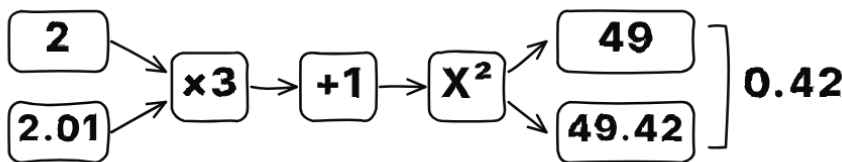
3. Describe the difference between the functional way of writing a layer, as in JAX, and the object way, as in PyTorch or TensorFlow.

Hint: Ask where the weights live in each style.

Answer: In the functional way a primitive is a function that receives its input and its weights side by side. In the object way the layer keeps its weights inside itself and is called on the input alone; that allows lazy initialization, creating the weights once the first input shows their size, but adds a layer of wrapping, and the two styles compute rates in different ways.

Three Ways to Find a Slope

Go back to the calculator chain from the start: two, times three, plus one, squared, which gave forty-nine. Suppose you did not know the chain rule and wanted the rate anyway. You could simply run the chain twice, once from two and once from two point zero one, and divide the difference of the answers by the nudge. The answers are forty-nine and about forty-nine point four two, so the rate comes out near forty-two, a little above it because the nudge is not infinitely small. It works, and it is the most honest method there is.



$$0.42 \div 0.01 \approx 42$$

✓ **CHECK ONLY**

...

2 RUNS PER WEIGHT

BILLIONS

This is numerical differentiation: apply the definition of a rate directly, nudge and measure. The trouble is the price. In a simple version, every single number you want a rate for costs two runs of the whole program. A chat assistant has billions of weights, so one round of training would need billions of runs of the entire network. That is why, as the book says, the method is useless for training and survives only as a check: you compute a few rates by nudging and compare them with what your clever method says.

The second way is the one you learned at school. Take a small function: $a \times \sin(x) + b \times x \times \sin(x)$. A program for algebra, such as SymPy in Python, can work out the formula of its rate exactly, the way you would on paper: $a \times \cos(x) + b \times \sin(x) + b \times x \times \cos(x)$. This is symbolic differentiation. At the number zero, with a and b both equal to one, the formula gives one: the cosine of zero is one, and the other two terms vanish.

$$f(x) = \underbrace{a \sin x} + \underbrace{b x \sin x}$$

SYMPY ↓

$$f'(x) = \underbrace{a \cos x} + \underbrace{b x \cos x} + b \sin x$$

$$X = 0, A = B = 1 \rightarrow 1$$

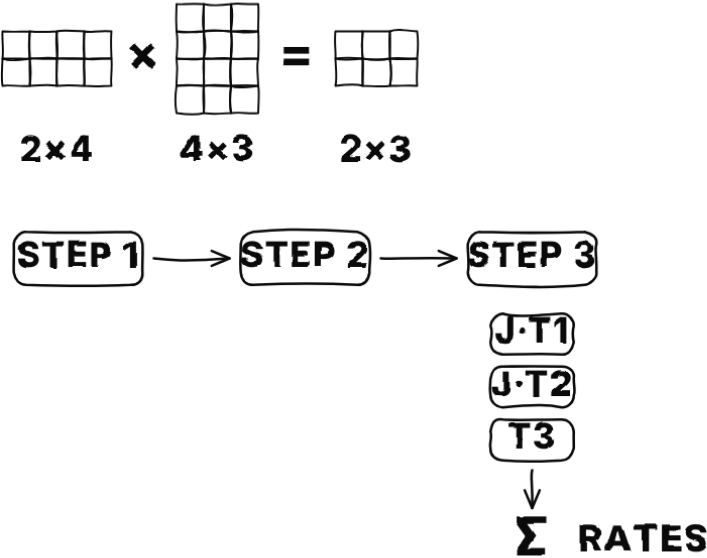
H = SIN X **KEEP ONCE**

HARD TO OPTIMISE

A Formula and Its Repeats

Look at the formula, though: the sine appears twice in the function and the cosine twice in the rate. A sensible program would compute each of them once, keep the value, and use it wherever it is needed. That sounds easy, but the book warns that finding the very best way to organise such a computation, so that nothing at all is calculated twice, belongs to a class of problems for which no fast general method is known; it is called optimal Jacobian accumulation. Automatic differentiation side-steps the search by following the structure of the program itself, step by step, and does far better than writing out the full formula; in fact it amounts to symbolic differentiation that is allowed to keep and reuse its intermediate results.

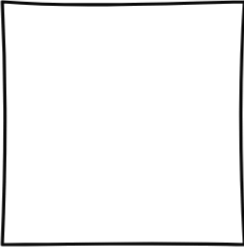
Now to the first of the two automatic methods, forward mode. It starts from the chain rule for two steps. Say a first primitive turns the input into an intermediate result, and a second turns that into the output. The table of rates of the output with respect to the input is the second step's table multiplied by the first step's table. The sizes must fit: if the input has three numbers, the intermediate result four and the output two, then the second table has two rows and four columns, the first has four rows and three columns, and their product has two rows and three columns, exactly one rate for every output against every input. Read the rule as a recipe: once you have run the first step and know its table, running the second step lets you update that table by multiplying it with the new one.



Forward mode turns this recipe into an algorithm. For each weight list it keeps a table called a tangent: how the current intermediate result moves with respect to those weights. At the first step nothing is stored yet, so the tangent of the first weights is simply the weight Jacobian of the first step. At the second step two things happen together: the first tangent is updated, multiplied by the input Jacobian of the second step, and a new tangent is started for the second weights, equal to the weight Jacobian of the second step. And so on down the chain: at each step, every tangent already stored is multiplied by that step's input Jacobian, the same one for all of them, and one new tangent is born from the step's weight Jacobian. At the very end the program adds up the numbers of the last result, and adding up the rows of each tangent gives the rates of the final score with respect to every weight.

The book then weighs the method. On the good side, all this work can be woven into the program as it runs, step by step, so the memory needed is roughly what the program itself needs. On the bad side, the heart of the algorithm, updating a tangent, multiplies two big tables, and that is very expensive. Suppose each step takes in and gives out a mini-batch table of thirty-two examples with a hundred features each, and one tangent belongs to a thousand weights. Unrolled, the input Jacobian is three thousand two hundred rows by three thousand two hundred columns, and the tangent is three thousand two hundred rows by a thousand columns. Their product costs about three thousand two hundred, times three thousand two hundred, times a thousand multiplications: some ten billion, for one update of one tangent at one step. The cost grows with the square of the batch size and the square of the number of features, so doubling both makes it sixteen times larger; for pictures, with their huge numbers of features, it soon becomes impossible.

INPUT J



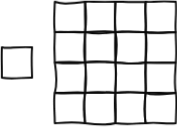
3,200 × 3,200

TANGENT



3,200 × 1,000

×



**x2 BATCH
x2 FEATURES
x16**

≈ 10,000,000,000

✓ **MEMORY: LOW**

The Price of One Tangent Update

Try it yourself

Try the honest method on a calculator with the square alone: square three, then square three point zero zero one. The answers are nine and about nine point zero zero six, so the rate is close to six, twice three, as the school rule says. Now imagine repeating those two runs for every one of the billions of weights of a chat assistant, and you will see why the honest method is kept only for checking.

Remember

Numerical differentiation nudges each number and reruns the program, so it costs two runs per weight and serves only to check other methods. Symbolic differentiation writes out the full formula of the rate, which repeats work unless intermediate values are kept, and organising that perfectly is a hard problem; automatic differentiation follows the program's own structure instead. Forward mode keeps one tangent per weight list, multiplies every stored tangent by each new step's input Jacobian, and adds a new tangent from the step's weight Jacobian. It needs little memory, but each update multiplies two large tables, a cost that grows with the square of batch size and of feature count, ten billion multiplications for one modest layer.

Questions

1. Why is numerical differentiation kept only as a check and not used for training?
 - a. Because it gives the wrong rate whenever the function contains a sine
 - b. Because it needs the full formula of the rate before it can start
 - c. Because each weight costs two runs of the whole program, and networks have billions of weights
 - d. Because it cannot be used on a calculator, only inside a framework

2. What makes the plain symbolic formula wasteful, and what does automatic differentiation do instead?
 - a. The formula repeats terms such as the sine and cosine; automatic differentiation follows the program's own steps and reuses intermediate results
 - b. The formula is always wrong at zero; automatic differentiation nudges the input instead
 - c. The formula needs a calculator; automatic differentiation needs only paper
 - d. The formula has too few terms; automatic differentiation adds the missing ones
3. In forward mode, what happens to the tangents already stored when the program runs a new step?
 - a. They are thrown away and rebuilt from the weight Jacobian of the new step
 - b. Each is multiplied by the new step's input Jacobian, the same one for all, and a new tangent is started
 - c. Only the most recent tangent is updated; the older ones stay as they were
 - d. They are added together into one tangent for the whole program
4. If both the batch size and the number of features are doubled, how does the cost of one tangent update in forward mode change?
 - a. It doubles
 - b. It grows four times

- c. It stays the same, because the number of weights is unchanged
- d. It grows sixteen times

Answers

1. **c)** Nudging and measuring works, but every number you want a rate for costs two runs of the whole program, and a chat assistant has billions of weights.
2. **a)** The sine appears twice in the function and the cosine twice in the rate; automatic differentiation follows the structure of the program and amounts to symbolic differentiation allowed to keep and reuse intermediate results.
3. **b)** At each step every tangent already stored is multiplied by that step's input Jacobian, shared by all of them, and one new tangent is born from the step's weight Jacobian.
4. **d)** The cost grows with the square of the batch size and the square of the number of features, so doubling both multiplies it by four and by four again.

Worked examples

The rate of the sine function The function is $a \times \sin(x) + b \times x \times \sin(x)$. Its rate, worked out symbolically, is $a \times \cos(x) + b \times x \times \cos(x) + b \times \sin(x)$. What is the rate at the number zero when a is two and b is one?

The first term is $a \times \cos(0)$; the cosine of zero is one, so it gives two.

The second term has the number itself as a factor, and the number is zero, so it gives zero.

The third term is $b \times \sin(0)$; the sine of zero is zero, so it gives zero.

Add the three terms.

Answer: Two.

The price of one tangent update A step takes in and gives out a mini-batch of thirty-two examples with a hundred features each, and a tangent belongs to a thousand weights. About how many multiplications does one update of that tangent cost?

Unrolled, the input and the output each have thirty-two times a hundred, three thousand two hundred numbers.

So the input Jacobian is three thousand two hundred by three thousand two hundred, and the tangent three thousand two hundred by a thousand.

Multiplying the two tables costs about the product of the three sizes: three thousand two hundred times three thousand two hundred times a thousand.

Three thousand two hundred squared is ten million two hundred and forty thousand; times a thousand gives about ten billion.

Answer: About ten billion multiplications, for one update of one tangent at one step.

Practice

1. A small model has a thousand weights. How many runs of the whole program does simple numerical differentiation need to get all their rates?

Hint: Each number you want a rate for costs a fixed number of runs.

Answer: Two thousand runs: two runs, with and without the nudge, for each of the thousand weights.

2. A first primitive turns five numbers into six, and a second turns those six into two. What are the shapes of the two tables in the chain rule, and of their product?

Hint: Each table has one row per output of its step and one column per input of its step.

Answer: The second step's table is two by six, the first step's is six by five, and their product is two by five: one rate for each of the two outputs against each of the five inputs.

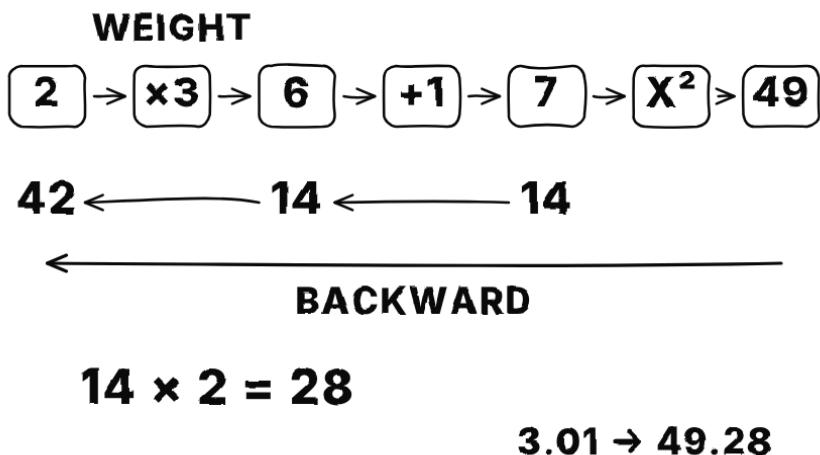
3. In forward mode, the program is running its fourth step. How many stored tangents are updated at this step, and how many new tangents are started?

Hint: Count the tangents the first three steps have left behind.

Answer: Three are updated, one for each earlier step, all multiplied by the fourth step's input Jacobian, and one new tangent is started from the fourth step's weight Jacobian.

Walking the Chain Backwards

Take the calculator chain once more: two, times three, plus one, squared, forty-nine. This time read it from the end. The last button, squaring at seven, magnifies a small change fourteen times, so the answer moves fourteen times as much as the value just before the square. One button earlier, adding one passes changes through unchanged, so the answer still moves fourteen times as much as the six that came out of the multiplication. One more button back, multiplying by three, and the answer moves forty-two times as much as the starting two. Walking backwards, you learned at every stop how much the final answer cares about the number standing there.

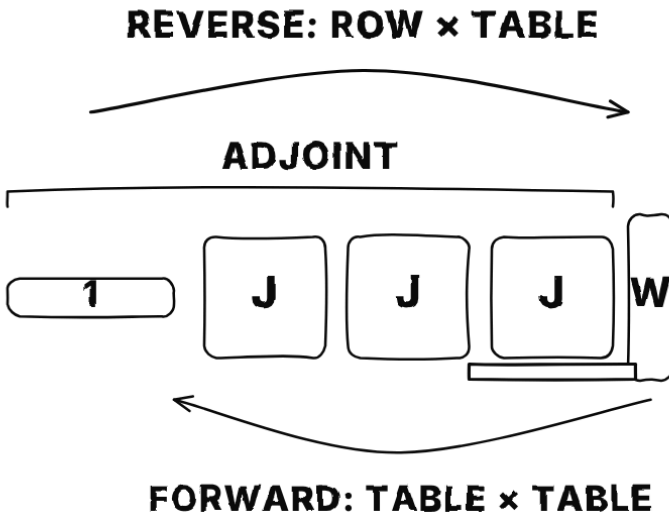


Reading the Chain From the End

Now pretend the three in the first button is an adjustable weight. How much does the answer move when that weight is nudged? The output of that button is the weight times two, so nudging the weight moves the output twice as much; and we already know the answer moves fourteen times as much as that output. Fourteen times two: the answer moves twenty-eight times as much as the weight. You can check it: with three point zero one instead of three, the chain gives about forty-nine point two eight. That small backward walk is the whole idea of this part.

The book writes out the rate of the final score with respect to the weights of one step, somewhere in the middle of the chain. It is a long product. First comes a row of ones, which stands for the final sum:

every number of the last result counts once in the score. Then come the input Jacobians of all the later steps, from the last one back to the step just after ours, and finally the weight Jacobian of our own step. The book gives a name to the first part of that product, the row of ones multiplied by all those later input Jacobians: it is the rate of the final score with respect to the intermediate result of our step.

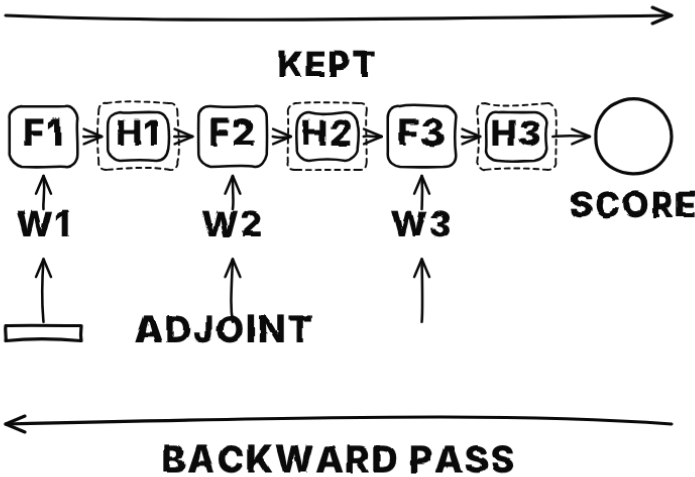


Which End to Start From

In a product of tables you may multiply in any order you like and get the same result. Forward mode multiplied from the step's own end outward, in the order the program runs, and paid for big table times table products. But look at the other end. It starts with a single row, and a row times a table is far cheaper than a table times a table; what comes out is again a single row. And the row for one step can be made

from the row of the step after it, by one more multiplication with an input Jacobian, walking backwards exactly as you did on the calculator.

That gives reverse-mode automatic differentiation. First run the whole program, from start to finish, and keep every intermediate result. Then start with the row of ones, and walk backwards from the last step to the first. At each step, multiply the current row by the step's weight Jacobian: that is the rate of the score with respect to that step's weights, done. Then multiply the row by the step's input Jacobian and carry the new row one step back.

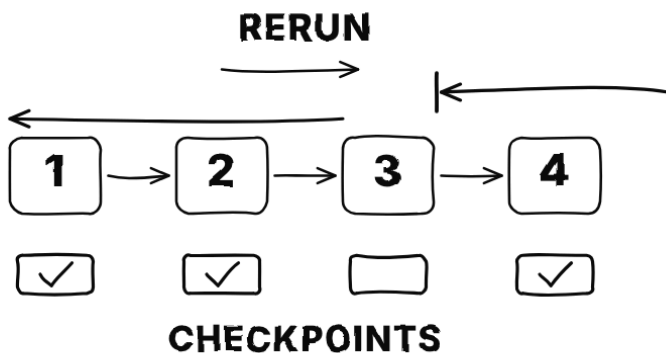


The Backward Pass

This backward walk is a second program, almost a mirror of the first, called the reverse or dual program. The rows it carries are called adjoints: each tells how the final score moves with every number of one intermediate result, while the tangents of forward mode told how an intermediate result moved with the weights. In the language of networks, running the original program is the forward pass, which is not the same thing as forward mode, and the backward walk is the backward pass.

The gain is large. Each backward step multiplies a single row by a table, a cost that grows only in proportion to the sizes involved. With the layer from before, thirty-two examples of a hundred features and a thousand weights, the two products of one backward step cost about ten million and about three million multiplications, roughly a thousand times less than the ten billion of one forward-mode update. This is what lets the photo app on your phone, or a chat assistant, be trained at all.

BACKWARD STEP $\approx$ 13 MILLION
FORWARD UPDATE $\approx$ 10 BILLION
 $\approx 1,000\times$ LESS



+25 % WORK

Cheap Steps, Costly Memory

The price is memory. The whole forward pass must finish before the backward pass can begin, and every intermediate result must be kept until the walk comes back to it. Gradient checkpointing trades some of that memory for extra work. The book's figure shows four blocks in a row: during the forward pass only the outputs of the first, second and fourth blocks are kept, as checkpoints. Walking back, the pass stalls at the third block, whose results were thrown away; so it reruns a short forward pass from the nearest checkpoint, the second block, rebuilds the missing results, and goes on. That costs a quarter more computation than a plain backward pass, and in general the fewer checkpoints are kept, the more must be computed again; training very large models leans on this trade every day.

Remember

The rate of the final score with respect to one step's weights is a long product that begins with a row of ones and runs through the later steps' input Jacobians to the step's weight Jacobian.

Reverse mode multiplies from the row end: it runs the whole program first, keeps every intermediate result, then walks backwards carrying a single row, the adjoint. At each step the adjoint times the weight Jacobian gives that step's rates, and the adjoint times the input Jacobian becomes the next adjoint back, as fourteen times two gave twenty-eight on the calculator. Row-times-table products make it about a thousand times cheaper than forward mode for one modest layer, at the cost of memory, which gradient checkpointing reduces by recomputing some results.

Questions

1. Why does reverse mode multiply the long product starting from the row of ones?
 - a. Because the row of ones is the largest table and should be used up first
 - b. Because a row times a table is far cheaper than a table times a table, and the result is again a single row
 - c. Because tables can only be multiplied in the order the program ran

- d. Because the row of ones removes the need for the weight Jacobians
2. What does an adjoint tell you?
- a. How an intermediate result moves when the weights of earlier steps are nudged
 - b. How many intermediate results must be stored during the forward pass
 - c. How the final score moves with every number of one intermediate result
 - d. How much memory a step needs during the backward pass
3. Why can the backward pass not start before the forward pass has finished?
- a. Because the backward walk begins at the last step and needs the intermediate results that the forward pass produces and keeps
 - b. Because the weights are only created at the end of the forward pass
 - c. Because the row of ones is computed by adding up all the weights
 - d. Because the framework must first translate the program into a formula
4. What happens when fewer checkpoints are kept?
- a. The backward pass becomes cheaper and needs less memory
 - b. The rates become less exact, because some results are guessed
 - c. The forward pass has to be skipped altogether

- d. Less memory is used, but more results must be computed again during the backward pass

Answers

1. **b)** The product may be taken in any order; starting from the row end, every multiplication is a single row times a table, which is cheap and gives a single row again.
2. **c)** The adjoints are the rows carried by the backward walk, each giving how the final score moves with the numbers of one intermediate result; the tangents of forward mode went the other way.
3. **a)** The walk starts from the end of the chain and uses every intermediate result on the way back, so the whole program must run first and keep them all.
4. **d)** Checkpointing trades memory for work: results that were thrown away are rebuilt by rerunning part of the forward pass, so fewer checkpoints mean more recomputation.

Worked examples

The cost of four blocks A network has four blocks. In the forward pass only the outputs of the first, second and fourth blocks are kept. How much more computation does this need compared with a plain backward pass, and why?

The forward pass runs the four blocks once each.

Walking back, the pass stalls at the third block, whose results were not kept.

It reruns the forward computation from the nearest checkpoint, the output of the second block, which means running the third block once more.

So five block runs are needed instead of four, one extra for every four.

Answer: A quarter more computation, one point two five times as much.

A backward walk with a weight A chain starts from one, multiplies by a weight equal to five, adds two, and squares, giving forty-nine. Walking backwards, how much does the answer move when the weight is nudged?

The square works on seven and magnifies a small change fourteen times: the answer moves fourteen times as much as the seven.

Adding two passes changes through: the answer moves fourteen times as much as the five coming out of the multiplication.

That five is the weight times the starting one, so nudging the weight moves it exactly as much: rate one.

Multiply the carried rate by this local rate: fourteen times one.

Answer: Fourteen times as much as the nudge of the weight.

Practice

1. A chain starts from three, multiplies by a weight equal to four, adds one, and squares. How much does the answer move compared with a small nudge of the weight?

Hint: Walk backwards from the square: first find how much the answer cares about the number entering the square.

Answer: Seventy-eight times: the square works on thirteen, so the rate there is twenty-six; adding one passes it through; the multiplication's output is the weight times three, so nudging the weight moves it three times as much; twenty-six times three is seventy-eight.

2. A network has eight blocks, and the outputs of all blocks except the fifth are kept as checkpoints. How much extra computation does the backward pass need?

Hint: Count how many block runs the forward pass makes, and how many must be repeated.

Answer: One eighth more: the forward pass runs eight blocks, and only the fifth must be rerun from the fourth block's checkpoint, so nine runs are needed instead of eight.

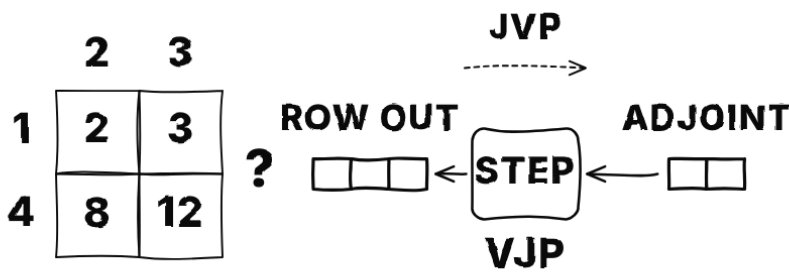
3. Explain in your own words why reverse mode is so much cheaper than forward mode, and what it pays for that.

Hint: Compare what is multiplied at each step in the two methods.

Answer: Forward mode multiplies big tables by big tables at every step, while reverse mode only ever multiplies one row by a table, whose cost grows just in proportion to the sizes; for a modest layer that is about a thousand times less work. The price is memory: the whole forward pass must be kept until the backward walk reaches each result, which checkpointing eases by recomputing some of them.

Never Build the Table

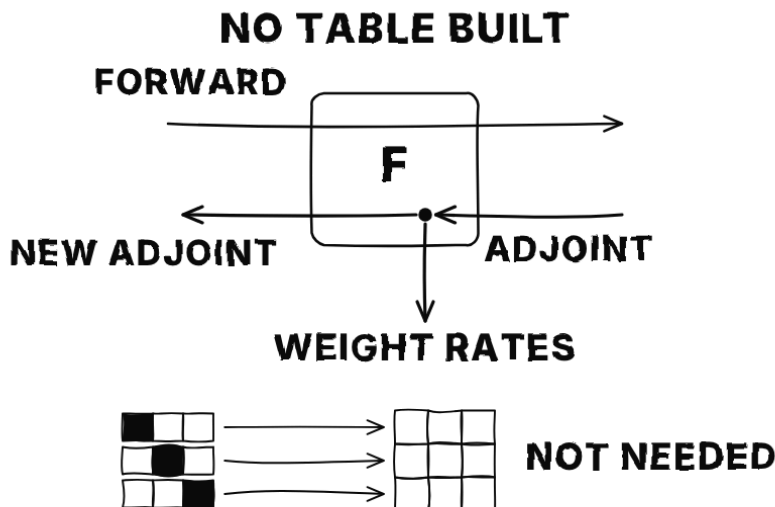
Draw a small times table on paper. Across the top write two numbers, two and three; down the left side write two others, one and four. Fill every box with the product of its row number and its column number: the first row reads two and three, the second eight and twelve. That little grid, which takes a child a minute, is exactly the set of rates a training program needs for the weights of a layer, and by the end of this part you will see why.



A Times Table and a Row In, Row Out

Look again at the backward walk of reverse mode. At each step only two things happen: the adjoint, a single row, is multiplied by the step's weight Jacobian, and the same row is multiplied by the step's input Jacobian. Nothing else is ever asked of a step. The book gives these two operations a name: vector-Jacobian products. A vector-Jacobian product is a function: you hand it a row with one number for each output of the step, and it hands back a row with one number for each input, or for each weight. Forward mode has its own mirror operation, the Jacobian-vector product, which for a function with a single output is the familiar rate along a chosen direction.

So a backward step is just two calls. Call the weight vector-Jacobian product with the adjoint, and out come the rates of the score with respect to that step's weights. Call the input vector-Jacobian product with the same adjoint, and out comes the new adjoint, ready to be carried one step back. The book's figure shows a primitive as a box: the forward pass goes in one side and out the other, and two backward arrows, one for each of these products, carry the adjoint back through it. And here is the point of the whole section: a complete system of automatic differentiation is a set of primitives, each paired with its two vector-Jacobian products, and no Jacobian ever has to be built in memory.

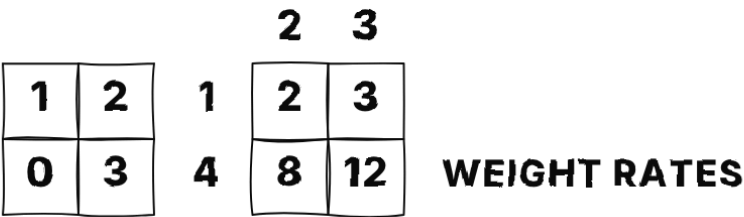


A Primitive With Two Backward Rules

Nothing is lost by this. If you really want a full Jacobian, you can call the product once with a row that has a one in the first place and zeros elsewhere, then with a one in the second place, and so on, and each call gives one row of the table. But the training program never wants the table; it only wants products with adjoints.

Now the payoff, for the two pieces a fully-connected layer is made of. First, a linear projection: the output is the weight table multiplied by the input. Its input Jacobian is just the weight table itself, but its weight Jacobian is a block with three directions, an output, a row of the weights and a column of the weights. The input product needs no block: each input number collects the adjoint entries of the outputs it fed, each weighted by the weight that connected them, which is the

same as passing the adjoint backwards through the weight table turned on its side. Take a weight table whose first row is one and two and whose second row is zero and three, and an adjoint of one and four. The first input fed the outputs through the weights one and zero, so it collects one times one plus four times zero, which is one; the second fed them through two and three, so it collects one times two plus four times three, which is fourteen.



INPUT RATES: 1, 14

The Two Products of a Linear Step

The weight product is even nicer, and it is your times table. The rate of the score with respect to the weight in a given row and column is the adjoint entry of that row times the input entry of that column. The book's reasoning in words: the score as seen by this step is a sum of products, each weight multiplied by its adjoint entry and by its input

entry, so nudging one weight moves the score by exactly its adjoint entry times its input entry. With inputs two and three and adjoint one and four, the grid is two and three, then eight and twelve: the table you drew at the start. So each time a linear projection runs forward, the backward pass sends the adjoint back through its turned weights and fills a times table for its weights, and no three-direction block appears anywhere.

Second, an activation function applied number by number, such as the ReLU, which has no weights, so only the input product matters. Each output depends only on the input in the same place, so the input Jacobian is a table that is zero everywhere except along its diagonal, where it holds the slope of the activation at each input. Multiplying a row by such a table simply scales each adjoint entry by the slope in the same place. For the ReLU with inputs minus one, two and three, the slopes are zero, one and one; an adjoint of five, six and seven comes back as zero, six and seven. For a layer of a thousand numbers the full table would hold a million entries, all but a thousand of them zero, and the product never needs it. This is how the photo app on your phone gets its rates: every operation in the training library carries its own small backward rule of this kind.

	SLOPES	ADJOINT
-1	0	5 → 0
2	1	6 → 6
3	1	7 → 7

1,000 × 1,000: MOSTLY ZEROS

The ReLU Only Scales

Remember

The backward walk only ever multiplies a row by a Jacobian, and these operations are the vector-Jacobian products, one for the input and one for the weights of each primitive. An automatic differentiation system is a set of primitives, each with its two products, and it never builds a Jacobian, though it could, one row at a time. For a linear projection the input product sends the adjoint back through the turned weight table, and the weight product is a times table of adjoint entries and input entries. For an activation applied number by number, the input product just scales each adjoint entry by the activation's slope at the matching input, as the ReLU turned five, six and seven into zero, six and seven.

Questions

1. Why does a system of automatic differentiation never need to build a Jacobian in memory?
 - a. Because the backward walk only ever multiplies a row by a Jacobian, and each primitive can do that directly with its two vector-Jacobian products
 - b. Because Jacobians are always zero except along their diagonal
 - c. Because the rates are found by nudging each weight instead
 - d. Because the forward pass already stores the Jacobians of every step

2. For a linear projection, what are the rates of the score with respect to its weights?
 - a. The weight table itself, turned on its side
 - b. The adjoint entries added together, once for every weight
 - c. A times table: the adjoint entry of the weight's row times the input entry of its column
 - d. A block with three directions that must be built before the backward pass
3. What does the input vector-Jacobian product of the ReLU do to the adjoint?
 - a. It adds the slope of the ReLU to every adjoint entry
 - b. It multiplies each adjoint entry by the ReLU's slope at the input in the same place
 - c. It replaces the adjoint with the inputs of the layer
 - d. It multiplies the adjoint by the full table of slopes, a million entries for a thousand numbers
4. If you really wanted the full input Jacobian of a primitive, how could you get it from its vector-Jacobian product?
 - a. By calling it once with a row of ones
 - b. By calling it with the weights instead of the adjoint
 - c. By running the forward pass twice with a nudge
 - d. By calling it with a row that has a one in one place and zeros elsewhere, once for each place, one table row per call

Answers

1. a) A backward step asks only for the adjoint times the weight Jacobian and the adjoint times the input Jacobian, so primitives paired with these

two products are enough.

2. **c)** The score as seen by the step is a sum of products of each weight with its adjoint entry and its input entry, so nudging one weight moves the score by that adjoint entry times that input entry.
3. **b)** The input Jacobian of an activation applied number by number is zero off its diagonal, so the product simply scales each adjoint entry by the slope in the same place, with no table built.
4. **d)** Each call with a row holding a single one gives one row of the table, so repeating it for every place rebuilds the whole Jacobian.

Worked examples

The weight rates of a projection A linear projection receives the inputs one and five, and the adjoint coming back is two and three.

What are the rates of the score with respect to its four weights?

The rate for the weight in a given row and column is the adjoint entry of the row times the input entry of the column.

First row, adjoint entry two: two times one is two, two times five is ten.

Second row, adjoint entry three: three times one is three, three times five is fifteen.

Set them out as a grid, rows by adjoint entries, columns by inputs.

Answer: The first row is two and ten, the second three and fifteen.

Back through a ReLU A ReLU layer received the inputs four, minus two, one half and minus three. The adjoint coming back is one, two, three and four. What adjoint does it pass on?

The ReLU's slope is one where the input is positive and zero where it is negative.

So the slopes are one, zero, one and zero.

Scale each adjoint entry by the slope in the same place.

Answer: One, zero, three and zero.

Practice

1. A linear projection has a weight table whose first row is two and one and whose second row is one and three. The adjoint coming back is one and two. What new adjoint does the input product pass on?

Hint: Each input collects the adjoint entries of the outputs it fed, each weighted by the weight that connected them; the first input fed the outputs through the first entry of each row.

Answer: Four and seven: the first input gets one times two plus two times one, which is four; the second gets one times one plus two times three, which is seven.

2. A primitive has five outputs and eight inputs. How many calls of its input vector-Jacobian product does it take to rebuild its whole input Jacobian, and how many numbers does each call give?

Hint: Each call with a single one gives one row of the table; ask how many rows the table has.

Answer: Five calls, one for each output, each giving a row of eight numbers, one per input.

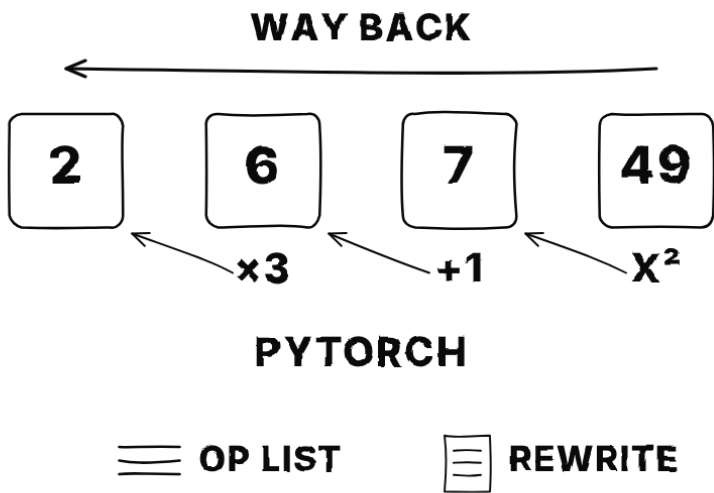
3. An activation function is applied to a layer of five hundred numbers. How many entries would its full input Jacobian hold, and how many of them would be zero?

Hint: The table has one row per output and one column per input, and only the diagonal is not zero.

Answer: Two hundred and fifty thousand entries, of which two hundred and forty-nine thousand five hundred are zero; only the five hundred on the diagonal hold slopes, which is why the product just scales the adjoint instead.

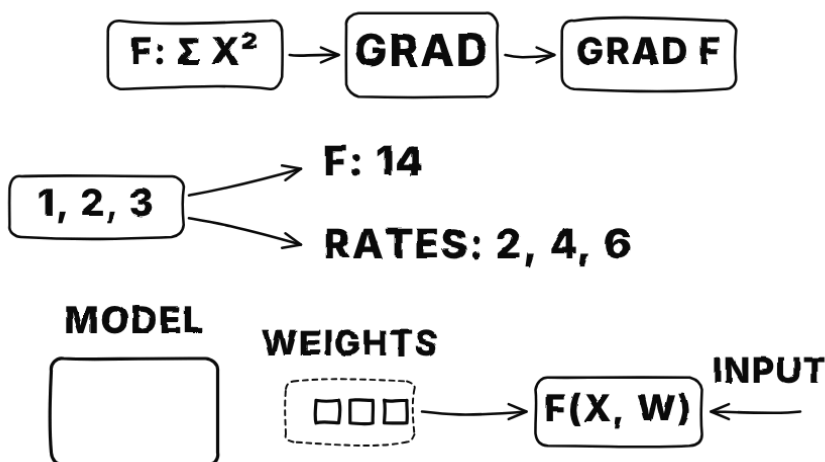
Notes Left Along the Way

Do the calculator chain one last time, but on sticky notes. Write the starting two on a note. Each time you press a button, write the result on a fresh note, and underneath it jot what made it: "times three, from the note before", "plus one, from the note before", "squared, from the note before". When you reach forty-nine, you can find your way back without remembering anything: each note tells you which operation made it and where it came from. That trail of notes is, almost literally, how PyTorch does reverse mode.



The book first says there are many ways to build a reverse-mode system, from keeping a running list of every operation performed, as TensorFlow does, to rewriting the program's own code into a second program that computes the rates. It then looks at the two styles we met earlier, functions and objects.

In a functional library such as JAX, everything is a function, and so is differentiation. A vector-Jacobian product becomes a function that makes functions. You give it a function and a point to look at; it gives you back a new function, which takes a row with one number per output and returns that row multiplied by the Jacobian at your point, one number per input. For a function with a single output there is something even simpler: give it the function, and it hands back a second function that computes the gradient, the rates with respect to every input, at any point you like. The book's example is the sum of the squares of ten numbers: the gradient function, given ten numbers, returns ten rates. Try it with three numbers, one, two and three: the sum of squares is fourteen, and the rates are two, four and six, twice each number, because each square grows twice as fast as its number. JAX supplies these function-making functions ready to use, PyTorch copies the same interface in a module of its own, and the result can be compiled into an optimised graph; combining the two kinds of products cleverly even gives second rates, the rates of rates.

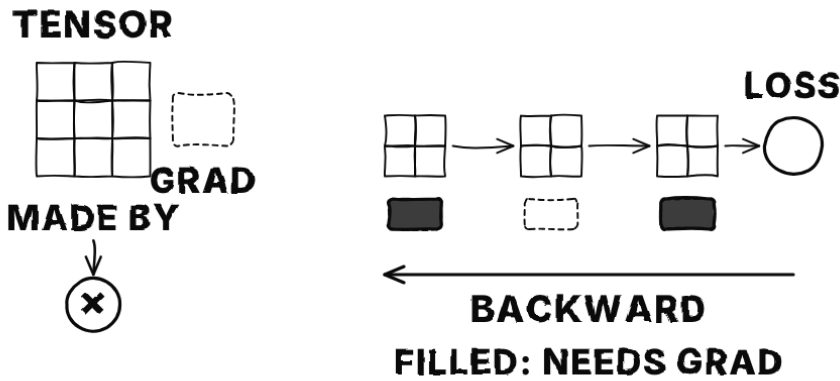


A Function That Makes Functions

Real models, though, are mostly written as objects that keep their weights inside. One way out is to purify them: pull the weights out of the object into a plain collection, and call the model as a function that receives its input and those weights side by side, which is what the functional style wants.

PyTorch also does the job directly, with the sticky notes. Every tensor, PyTorch's name for a table of numbers, carries two extra things: an empty slot for its gradient, and a note of the operation that produced it. The book's figure shows this on the left: a tensor, its empty gradient slot, and an arrow to the operation it came from. When you ask for the backward pass on a single number, such as the loss of a batch of photos, PyTorch follows the notes backwards through the

computational graph, applying each operation's backward rule as it goes. On the right of the figure the slots fill up: the gradients are stored inside the tensors themselves, but only in those whose owner has said that they need a gradient, so that no memory is wasted on the rest.



MICROGRAD

Tensors Carry Their Own Notes

The book admits this is only a sketch and points to the official documentation for the details. Its warmest advice is to build a small reverse-mode system yourself, and it names one of the many teaching versions online, micrograd, which fits in a few short files.

Try it yourself

Write the chain two, times three, plus one, squared on sticky notes, each note saying what made it. Now start from the last note and walk back, writing on each note how much the final forty-nine moves with it: fourteen on the seven, fourteen on the six, forty-two on the two. You have just done by hand what PyTorch does when you ask it for the backward pass.

Remember

Reverse mode can be built in many ways, from a running list of operations to rewriting the program's code. In the functional style of JAX, differentiation is a function that makes functions: given a function and a point, it returns the vector-Jacobian product, and for a single output it returns the gradient function, as the sum of squares of one, two and three has rates two, four and six. Objects that keep their weights inside can be purified into such functions. PyTorch instead lets each tensor carry a note of the operation that made it and an empty gradient slot, walks the notes backwards on request, and fills the slots of the tensors that asked for gradients.

Questions

1. In the functional style of JAX, what do you get back when you ask for the vector-Jacobian product of a function at a point?

- a. The full Jacobian of the function at that point, as a table
 - b. A new function that takes a row with one number per output and returns that row times the Jacobian
 - c. The value of the function at that point, computed twice
 - d. A list of the operations the function performed, for later use
2. How does PyTorch know how to walk backwards through a computation?
- a. It reruns the forward pass with a nudge on every weight
 - b. It rewrites the model's code into a second program before training starts
 - c. Each tensor carries a note of the operation that produced it, and the backward pass follows these notes
 - d. It asks the model object for a formula of its rates
3. Why does PyTorch fill the gradient slot only of tensors that have asked for a gradient?
- a. So that no memory is wasted on gradients nobody needs
 - b. Because other tensors have no operation that produced them
 - c. Because the backward pass cannot pass through tensors without a gradient
 - d. So that the forward pass can run faster
4. What does purifying a model mean?
- a. Removing the activation functions so that the model becomes linear
 - b. Deleting the weights that have a gradient of zero
 - c. Compiling the model into an optimised graph

- d. Pulling the weights out of the object so that the model can be called as a function of its input and its weights

Answers

1. **b)** Differentiation there is a function that makes functions: given a function and a point, it returns a new function that multiplies any row by the Jacobian at that point.
2. **c)** Every tensor keeps a note of the operation that made it, so asking for the backward pass on a single number lets PyTorch follow the notes back through the graph.
3. **a)** The gradients are stored inside the tensors themselves, and only where the owner said they are needed, so that no memory is wasted on the rest.
4. **d)** Objects keep their weights inside; `purifying` pulls them out into a plain collection, so the model can be called the functional way, input and weights side by side.

Worked examples

The gradient of a sum of squares The book's example asks for the gradient function of the sum of the squares of ten numbers. How many numbers does that gradient function return, and what does it return for the three numbers three, minus one and two, if we use three numbers instead of ten?

The function has a single output, so its gradient has one rate for each input.

With ten numbers in, the gradient function returns ten rates.

Each square grows twice as fast as its number, and the other squares do not change when one number is nudged.

So each rate is twice its number: six, minus two and four.

Answer: Ten numbers for ten inputs; for three, minus one and two, the rates are six, minus two and four.

Practice

1. Using the gradient function of the sum of squares, what are the rates at the numbers one half and four?

Hint: How fast does a square grow compared with its number?

Answer: One and eight: each rate is twice its number.

2. A function takes six numbers in and gives two numbers out. You ask the functional library for its vector-Jacobian product at some point. How many numbers does the function you get back take, and how many does it return?

Hint: The row it takes has one number per output of the original function.

Answer: It takes two numbers, one per output, and returns six, one per input: the row multiplied by the two-by-six Jacobian at that point.

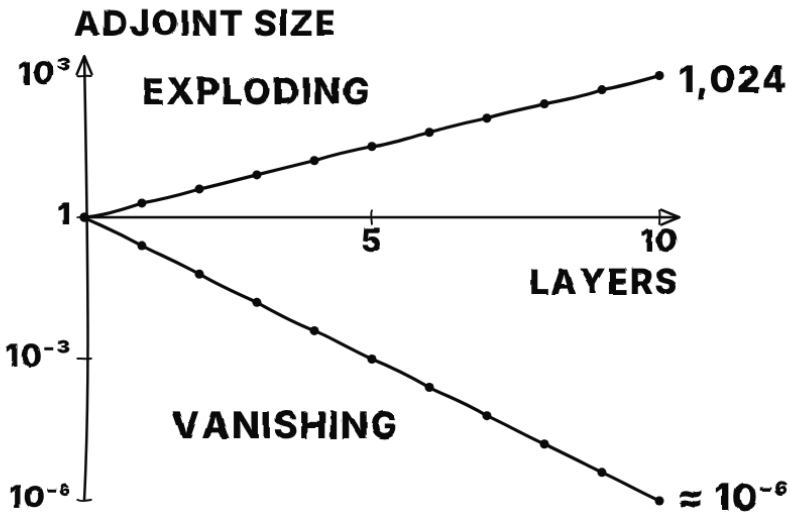
3. Explain in your own words how a trail of sticky notes lets you do the backward pass without remembering the program.

Hint: Think of what each note says about where its number came from.

Answer: Each note records its value, the operation that made it and the note it came from. Starting from the final result you can follow these links backwards and, at each note, apply that operation's backward rule to find how much the final result moves with it. PyTorch does the same with the notes it attaches to every tensor.

A Quarter, Ten Times Over

Type one into a calculator and multiply by a quarter, ten times in a row. You might expect something small, but not this small: the display shows about one millionth. Now clear it, type one again, and multiply by two, thirty times in a row: the display passes a billion. These two runs of buttons are the whole of this part.



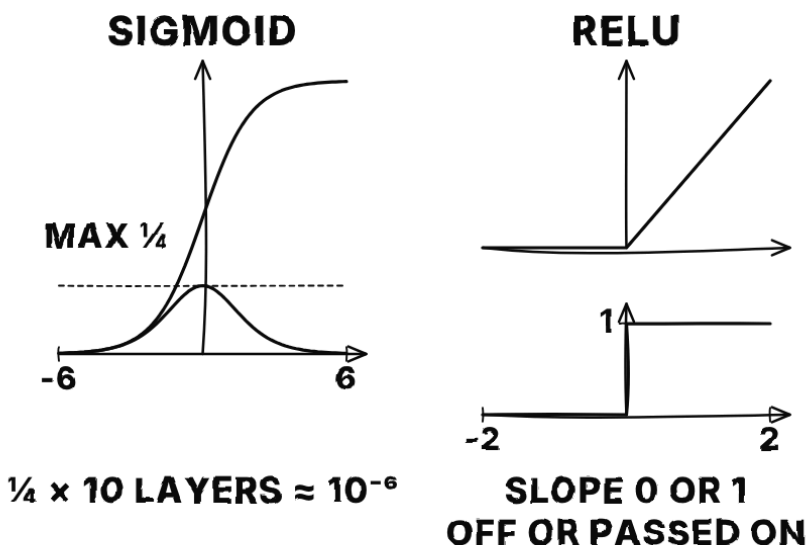
Shrinking and Growing, Layer by Layer

Recall what an activation function does in the backward pass: each adjoint entry is multiplied by the activation's slope at the matching input. A deep network has an activation in every layer, so on its way

back the adjoint is multiplied by such a slope again and again, once per layer. If the slopes are below one everywhere, the adjoint shrinks with every layer, as your quarter did, and after enough layers the rates for the early weights are practically zero: this is the vanishing gradient problem. If the slopes are above one everywhere, the adjoint grows with every layer, as your doubling did, and the rates race off towards infinity: the exploding gradient problem.

These are not just curiosities, because a computer stores numbers with limited precision, usually thirty-two binary digits or fewer. A number that shrinks far enough is simply rounded to zero, and one that grows far enough no longer fits, and with many layers this happens quickly. The book adds a curious twist: because of these tiny rounding effects, a stack of purely linear layers on a computer is not perfectly linear, and people have even used this to train deep networks with no activations at all.

The sigmoid shows how vanishing gradients appear. It was a common activation in the past, a soft version of a step: an S-shaped curve that rises smoothly from zero to one. Its slope at any point equals its value times one minus its value. Since the value lies between zero and one, that product is largest when the value is one half, and then it is a half times a half, a quarter; at a value of nine tenths it is only nine hundredths. The book's figure shows the S-curve with its slope beside it, a low bump that never rises above a quarter. So every sigmoid layer cuts the adjoint at least to a quarter, and ten of them cut it to about a millionth, your first calculator run.



Sigmoid Slopes Versus ReLU Slopes

Could an activation have a slope of exactly one everywhere? Only a function that does nothing at all, returning its input unchanged, and then the layers around it melt into one linear layer. We need something linear enough to keep the slopes harmless, yet bent enough to keep the layers apart. The ReLU fits: its slope is zero for negative inputs and one for positive ones. Its figure shows a line lying flat along zero and then rising at forty-five degrees, and beside it the slope, a step from zero up to one. So each adjoint entry is either switched off, which leaves many zeros and saves work, or passed on unchanged, so nothing shrinks or explodes.

There is a bonus for memory. The ReLU keeps positive numbers and turns negative ones into zero, so its output tells you just as well as its

input where the slope is one and where it is zero. The backward pass can therefore work from the output, and the input need not be kept: the layer may overwrite it in the forward pass, saving a little memory without spoiling any rate, and PyTorch offers exactly this as an in-place option.

Remember

Every activation multiplies the adjoint by its slope on the way back, so in a deep network these slopes multiply together. Slopes below one make the rates of early layers vanish, slopes above one make them explode, and limited precision on computers turns both into real failures. The sigmoid's slope is never more than a quarter, so ten sigmoid layers can shrink the adjoint to a millionth. The ReLU's slope is zero or one, so the adjoint is either switched off or passed on unchanged, and since its output shows where the slope is one, its input can be overwritten to save memory.

Questions

1. Why does a deep network with sigmoid activations risk vanishing gradients?
 - a. Because the sigmoid's output can be larger than one
 - b. Because the sigmoid's slope is never more than a quarter, and these slopes multiply layer after layer
 - c. Because the sigmoid switches off every negative input

- d. Because the sigmoid has no slope at zero
2. Why are vanishing and exploding gradients real failures and not just very small or very large numbers?
- a. Because the rates then point in the wrong direction
 - b. Because the loss stops being a single number
 - c. Because computers store numbers with limited precision, so tiny numbers become zero and huge ones no longer fit
 - d. Because the forward pass then has to be run twice
3. Why not use an activation whose slope is exactly one everywhere?
- a. Because only the function that returns its input unchanged has that slope, and then the layers around it melt into one linear layer
 - b. Because such a function would make every gradient explode
 - c. Because its slope would be zero for negative inputs
 - d. Because computers cannot store the number one exactly
4. Why may a ReLU layer overwrite its input during the forward pass?
- a. Because the ReLU has no backward rule at all
 - b. Because the input is never needed again by any layer
 - c. Because overwriting makes the rates more exact
 - d. Because its output shows just as well as its input where the slope is one and where it is zero

Answers

1. **b)** On the way back the adjoint is multiplied by the activation's slope in every layer, and the sigmoid's slope is at most a quarter, so ten layers can shrink it to about a millionth.
2. **c)** Numbers are stored with thirty-two binary digits or fewer, so a number that shrinks far enough is rounded to zero and one that grows far enough no longer fits.
3. **a)** The only function with slope one everywhere does nothing at all, and without a bend the linear layers merge into one; the ReLU is linear enough yet still bent.
4. **d)** The ReLU keeps positive numbers and zeroes negative ones, so the backward pass can read the slopes from the output, and the input can be overwritten to save memory.

Worked examples

The sigmoid's largest slope The sigmoid's slope equals its value times one minus its value. What is the slope when the value is one half, and when it is eight tenths? And by how much can five sigmoid layers at most scale an adjoint entry?

At the value one half: a half times a half is a quarter.

At the value eight tenths: eight tenths times two tenths is sixteen hundredths, less than a quarter.

The slope is largest at one half, so it never exceeds a quarter.

Through five layers the adjoint entry is multiplied by at most a quarter five times: a quarter of a quarter of a quarter of a quarter of a quarter is one part in one thousand and twenty-four.

Answer: A quarter and sixteen hundredths; five layers scale the adjoint entry to at most about one thousandth.

Practice

1. What is the sigmoid's slope when its value is two tenths, and when it is ninety-nine hundredths?

Hint: Multiply the value by one minus the value.

Answer: Sixteen hundredths, and about one hundredth: two tenths times eight tenths, and ninety-nine hundredths times one hundredth, which is ninety-nine ten-thousandths.

2. An adjoint entry of one passes back through three layers whose activation slopes are each one and a half. What does it become? And what would it become through three ReLU layers whose inputs were all positive?

Hint: On the way back each layer multiplies the adjoint entry by its slope.

Answer: Three and three eighths, since one and a half three times over is three point three seven five, the start of an explosion; through three ReLU layers with positive inputs it stays one, because each slope is one.

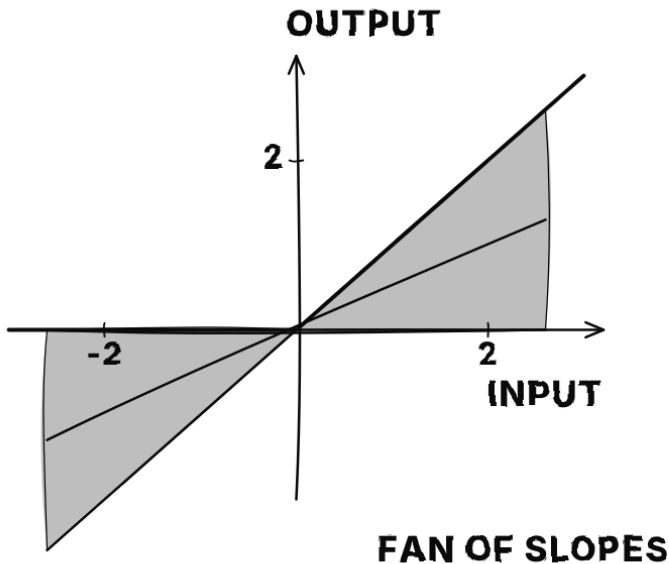
3. Explain in your own words why the ReLU avoids both vanishing and exploding gradients for the entries it lets through.

Hint: What are the only two slopes the ReLU can have?

Answer: The ReLU's slope is zero for negative inputs and one for positive ones. An entry is either switched off or multiplied by exactly one, so the entries that pass are never shrunk or enlarged, however many layers there are.

A Ruler at the Corner

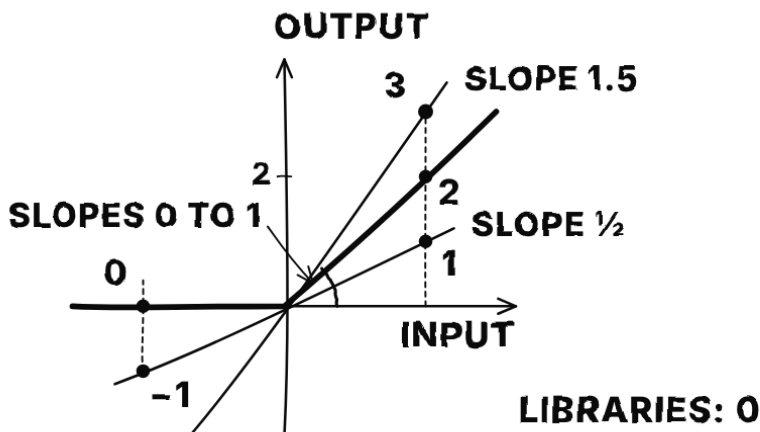
Draw the ReLU on a sheet of paper: a line lying flat along the horizontal axis on the left, then, from the corner at zero, a line rising at forty-five degrees to the right. Now lay a pencil through the corner and tilt it slowly. Held flat, it stays under the drawing everywhere; tilted up to forty-five degrees, it still stays under; tilted any steeper, it cuts through the rising arm, and tilted downwards, it cuts through the flat one on the left. So at the corner there is not one slope but a whole fan of them, every tilt from flat to forty-five degrees.



A Pencil Tilted at the Corner

This is the small detail the book has put off until now. The ReLU has no ordinary slope at zero, so a network built from ReLUs is not smooth everywhere, and the rules of the earlier parts seem to break at that point. The practical answer is that it hardly ever matters. Training starts from random weights and moves in small random steps, so the chance that an input lands exactly on zero is practically nil, and a hair to either side the slope is perfectly well defined.

The more careful answer uses the fan you just drew. For a function that curves upwards, a convex function, a subgradient at a point is the slope of a line through that point which stays below the whole function everywhere. Where the function is smooth, only one such line exists, the ordinary tangent, and its slope is the ordinary rate. At a corner many such lines exist, and the collection of their slopes is called the subdifferential. For the ReLU, the subdifferential is zero for negative inputs, one for positive inputs, and at zero it is every number from zero to one. Check one of them: take the line through the corner with slope one half. At the input two it gives one, while the ReLU gives two, so the line is below; at minus two it gives minus one, while the ReLU gives zero, so it is below again. A slope of one and a half fails: at the input two it gives three, above the ReLU's two. In practice most libraries simply choose zero as the ReLU's slope at zero, and following one chosen subgradient at every step of training is called subgradient descent.

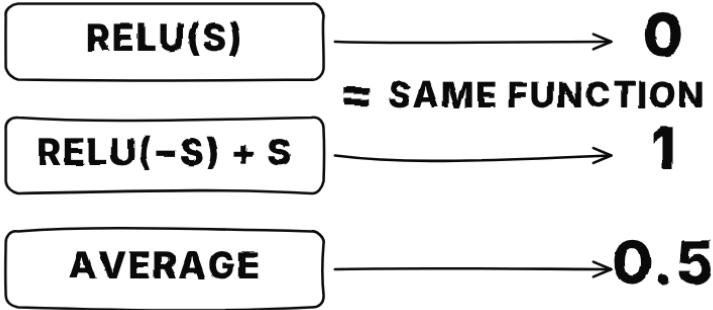


Checking a Subgradient

The book admits the story is trickier still. For functions that do not curve upwards, a subgradient may not exist at all, and mathematicians use looser versions that only look at a small neighbourhood of the point, such as the Clarke subdifferential. And automatic differentiation itself can trip here, because two ways of writing the same function may give different slopes at the corner. The book's example: take the ReLU of minus the number and add the number back; for any number this gives exactly the ReLU again. Average that with the plain ReLU and you still have the ReLU. Yet at zero PyTorch's backward pass gives zero for the first, one for the second and one half for the third: three versions of the same function, three answers, which is why proving such programs correct needs more refined chain rules.

○ CLARKE

SLOPE AT 0



One Function, Three Answers

The chapter ends by sending you to practise. Rebuild a small reverse-mode system, for instance by studying micrograd, which works on single numbers; once the graph branches and merges, you must first put its steps in an order that respects who feeds whom, a cheap sorting step, or the backward pass will follow wrong paths. Then write a primitive of your own in PyTorch, giving its forward computation and its vector-Jacobian products, for example one of the trainable activations of the previous chapter, even though PyTorch could work out its backward pass alone. In JAX the same steps exist: a teaching version of JAX called autodidax, custom backward rules for new primitives, and a cookbook of advanced uses such as second rates.

Here the chapter closes. From a calculator chain and its forty-two, you have followed the rates of a training program forwards and backwards, seen why the backward walk wins, why it never builds its tables, and how the ReLU's harmless slopes and even its one sharp corner are handled.

Remember

The ReLU has a corner at zero, but in training an input almost never lands exactly there, so the ordinary slope is nearly always defined. A subgradient of a convex function is the slope of a line through a point that stays below the whole function; at a corner the set of them, the subdifferential, for the ReLU at zero is every slope from zero to one, and libraries usually pick zero. For other functions subgradients need not exist, and equivalent ways of writing the ReLU make PyTorch return zero, one or one half at zero, so the corner needs care in proofs. The book closes by urging practice: rebuild a small system such as micrograd, ordering the steps of a branching graph, and write primitives with their own vector-Jacobian products in PyTorch or JAX.

Questions

1. Why does the ReLU's corner at zero hardly ever matter in practice?
 - a. Because libraries remove all zeros from the inputs before training

- b. Because training starts from random weights and moves in small random steps, so an input almost never lands exactly on zero
 - c. Because the ReLU's slope at zero is always one
 - d. Because the backward pass skips the ReLU layers entirely
2. Which slopes are subgradients of the ReLU at zero?
- a. Only zero, the value libraries choose
 - b. Only one half, halfway between the two arms
 - c. Every slope from zero to one, since those lines through the corner stay below the whole ReLU
 - d. Every slope from minus one to one
3. What does the book's example of three ways of writing the ReLU show?
- a. That equivalent ways of writing the same function can make automatic differentiation return different slopes at the corner
 - b. That two of the three ways are not really the ReLU
 - c. That PyTorch computes the ReLU wrongly for positive inputs
 - d. That subgradients do not exist for the ReLU
4. When the graph of a program branches and merges, what must a small reverse-mode system such as micrograd do before the backward pass?
- a. Replace every ReLU by a sigmoid
 - b. Run the forward pass twice to check the results
 - c. Build the full Jacobian of every step

- d. Put the steps in an order that respects who feeds whom, a cheap sorting step

Answers

1. **b)** With random starting weights and small random steps, the chance of an input landing exactly on zero is practically nil, and a hair to either side the slope is well defined.
2. **c)** A line through the corner stays below the ReLU everywhere exactly when its slope is between flat and forty-five degrees, so the subdifferential at zero is every number from zero to one.
3. **a)** All three are the same function, yet PyTorch's backward pass gives zero, one and one half at zero, so the corner needs refined chain rules in proofs.
4. **d)** Once the graph branches and merges, its steps must be ordered so that the backward pass does not follow wrong paths; micrograd does this with a cheap sort.

Worked examples

Three ReLUs, three slopes The second version is the ReLU of minus the number, plus the number; the third is the average of the plain ReLU and the second version. Check that the second version equals the ReLU at three and at minus three, and find the slopes PyTorch's backward pass gives at zero, knowing that it takes the plain ReLU's slope at zero to be zero.

At three: the ReLU of minus three is zero, plus three gives three, the same as the ReLU of three.

At minus three: the ReLU of three is three, plus minus three gives zero, the same as the ReLU of minus three.

At zero, the plain ReLU's slope is taken as zero.

The second version's slope is minus the ReLU's slope, because of the minus sign inside, plus one for the added number: minus zero plus one is one.

The third version is the average of the two, so its slope is the average of zero and one.

Answer: The second version equals the ReLU at both points; the slopes at zero are zero, one and one half.

Testing a subgradient Is a slope of one quarter a subgradient of the ReLU at zero? Test the line through the corner with that slope at the inputs four and minus four.

At four, the line gives one quarter of four, which is one; the ReLU gives four, so the line is below.

At minus four, the line gives minus one; the ReLU gives zero, so the line is below again.

A line through the corner with a slope between zero and one stays below on both arms, so it is below everywhere.

Answer: Yes, one quarter is a subgradient of the ReLU at zero.

Practice

1. Is a slope of eight tenths a subgradient of the ReLU at zero? Check the line through the corner at the inputs five and minus five.

Hint: Compare the line's value with the ReLU's value at each input.

Answer: Yes: at five the line gives four, below the ReLU's five; at minus five it gives minus four, below the ReLU's zero; and eight tenths lies between zero and one.

2. Is a slope of minus one half a subgradient of the ReLU at zero? Find an input where the line through the corner rises above the ReLU.

Hint: Look on the left, where the ReLU lies flat along zero.

Answer: No: at minus two the line gives one, above the ReLU's zero, so it does not stay below the whole function.

3. Explain in your own words what a subgradient is, and why a smooth point has only one.

Hint: Think of a pencil laid through the point and tilted.

Answer: For a function that curves upwards, a subgradient at a point is the slope of a line through that point that stays below the whole function. At a smooth point only the tangent does this, so the only subgradient is the ordinary slope; at a corner the pencil can tilt through a whole fan of positions, giving many subgradients.