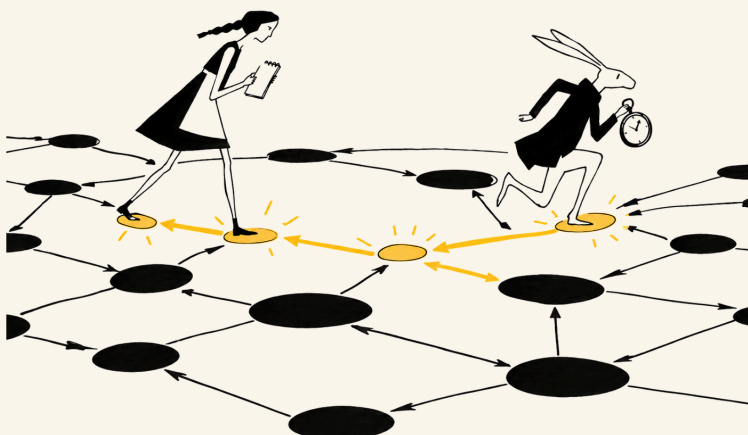


Alice's Adventures in a Differentiable Wonderland



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 Recipe Made of Small Steps

Why the Obvious Ways Are Too Slow

Walking Back From the Score

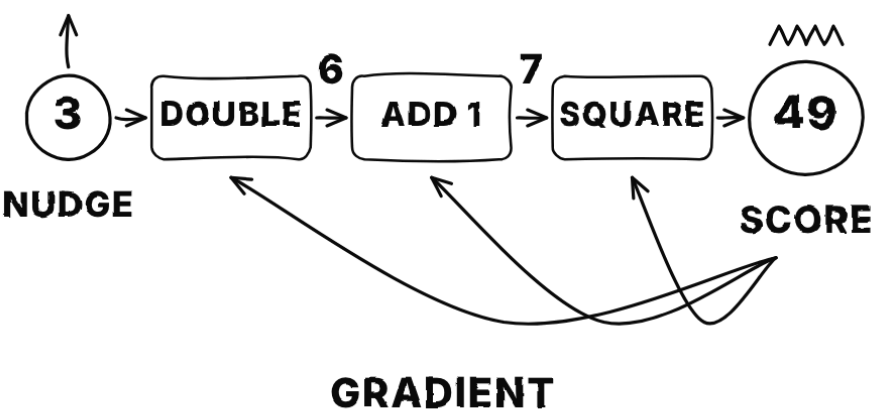
Asking the Table One Question

How Frameworks Walk Back, and Why ReLU Won

The Corner of the ReLU

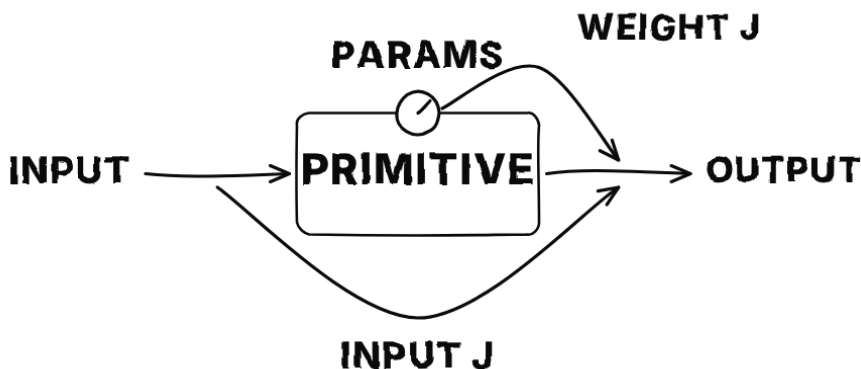
A Recipe Made of Small Steps

Take a calculator and a scrap of paper, and do a little sum in stages. Start with the number three, double it, add one, then square the result. You get forty-nine, and every stage was a tiny step you know by heart. Now ask a harder question: if you nudge the starting three up a hair, how much does the forty-nine move, and which of the stages is to blame for how much? That question, asked of a much longer recipe, is what this chapter is about.



When a chat assistant or a phone's photo app is trained, the program behind it is such a recipe, only with millions of stages and millions of adjustable numbers. At the end of the recipe sits a single score that says how badly the model did on a batch of examples; the book calls it the loss. Training means turning each adjustable number a little in the direction that lowers that score, and for that we need to know, for every number, how strongly the score reacts to it. That collection of reactions is the gradient, and working it out by machine, for any program at all, is called automatic differentiation. The method this chapter builds is the one every modern framework uses; neural network people call it back-propagation, and computer scientists call it reverse-mode automatic differentiation.

The author asks us to picture a program as the list of small operations it actually performs when it runs, one after another. The book calls each small operation a primitive, and the whole chain of them a computational graph. A primitive takes something in, uses some adjustable numbers of its own, the parameters, and hands something out. What counts as one primitive is up to us: a single multiplication of tables of numbers, a whole layer of a network, or even a complete smaller model can be treated as one step. This is the same habit programmers have everywhere, building big things from blocks that are themselves built from blocks.

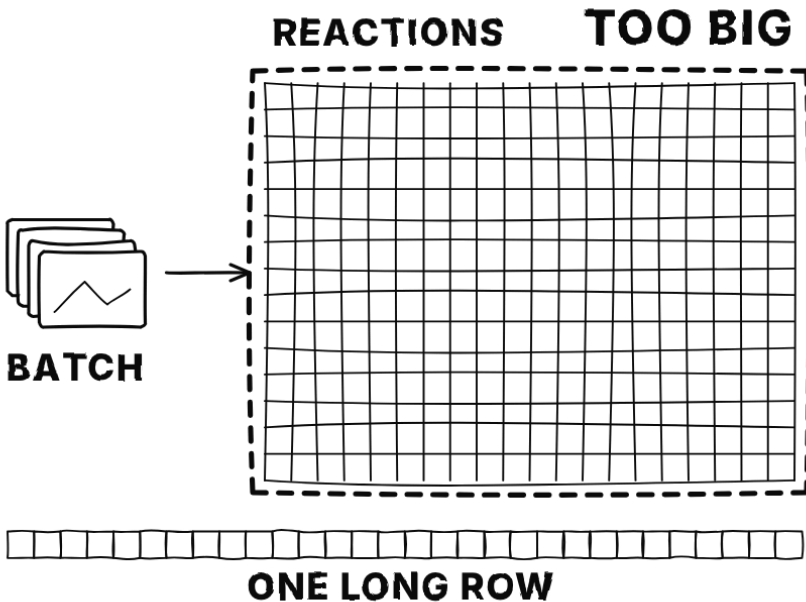


One Primitive, Two Reactions

The book asks only one thing of each primitive. We must know how its output reacts when its input is nudged, and how its output reacts when its own parameters are nudged. These two tables of reactions are called the input Jacobian and the weight Jacobian, and they are the only local knowledge the whole method needs. At first sight the problem then looks finished, because school calculus already has the chain rule: the reaction of a chain of steps is the reactions of the single steps multiplied together. The catch, and the real subject of the chapter, is doing that multiplication cheaply enough for a model the size of a photo app's network.

Some steps have a sharp corner where the slope is not defined, such as the popular ReLU, which lets positive numbers through and turns

negative ones into zero. The book promises to deal with such corners later in the chapter; truly jumpy operations, like drawing a random sample, are left for its second volume.



A Table Too Big to Keep

Here is a first hint of why cheapness matters. A photo app never processes one picture at a time during training; it takes a batch, say a few dozen pictures, each described by thousands of numbers. The table of reactions of even one layer then has to connect every number of every picture going in with every number of every picture coming out. The book keeps things simple by laying all of those numbers out in one long row, so that "the size of the input" always means everything at once, batch included. Counted that way, such a table becomes so enormous that we may know perfectly well how to

compute it and still not want to store it in memory at all. Hold on to that thought; it is the key to the clever method that comes later.

Finally, the author notes that frameworks write these steps down in two styles. In JAX, everything is a function: a step is handed its input and its parameters, exactly as the book writes it. In PyTorch or TensorFlow, a layer is an object that keeps its own parameters inside it, like an appliance with its settings built in, and you simply switch it on with the input. The appliance style is handy, for instance because the layer can wait to create its parameters until it first sees how big its input is, but it adds a small layer of translation between the book's way of writing and real code. That difference in style shows up again in how the two kinds of framework compute gradients.

Remember

Training an AI model means nudging millions of adjustable numbers to lower one final score, and automatic differentiation is the machine method for finding how that score reacts to each number. A program is seen as a chain of small steps called primitives, each with an input, its own parameters and an output. For each step we only need to know how its output reacts to its input and to its parameters: its input Jacobian and its weight Jacobian. The chain rule says how to combine those reactions, but the tables involved can be so huge that the real challenge is combining them cheaply. JAX writes steps as plain functions, while PyTorch packs each layer with its parameters into an object, and this shapes how each computes gradients.

Questions

1. Why does training a model need to know how the final score reacts to every adjustable number?
 - a. Because each number is turned a little in the direction that lowers the score
 - b. Because the score must be printed for every number separately
 - c. Because numbers that do not affect the score are deleted before training
 - d. Because the chain rule only works when every number is known
2. What does the method ask to know about each single step of the program?
 - a. Only the step's output for a few test inputs
 - b. How the step's output reacts to nudges of its input and of its own parameters
 - c. How long the step takes to run on a phone
 - d. How the final score reacts to the step, worked out in advance
3. If the chain rule already says how to combine the reactions of the steps, what is left to solve?
 - a. Finding a new rule, because the chain rule fails for long programs
 - b. Deciding which steps count as primitives, since only one choice is correct

- c. Doing the combining cheaply, because the tables of reactions can be enormous
 - d. Removing the steps with sharp corners such as the ReLU
4. How does a PyTorch-style layer differ from a step written the JAX way?
- a. The PyTorch layer has no parameters at all
 - b. The JAX step keeps its parameters hidden inside an object
 - c. The PyTorch layer keeps its parameters inside itself, while the JAX step is a function handed both input and parameters
 - d. There is no difference; the two names describe the same style

Answers

- 1. a)** Training turns each adjustable number a little in the direction that lowers the score, and for that the reaction of the score to every number, the gradient, is needed.
- 2. b)** For each primitive we only need its input Jacobian and its weight Jacobian, the reactions of its output to its input and to its parameters; everything else is built from these.
- 3. c)** The chain rule settles what to compute; the real challenge of the chapter is doing that multiplication cheaply enough for a model the size of a photo app's network.
- 4. c)** In JAX everything is a function given its input and its parameters, while in PyTorch a layer is an object with its parameters built in, like an appliance you switch on with the input.

Worked examples

Who is to blame in a short recipe Start with 3, double it, add 1, then square the result, which gives 49. If the starting 3 is nudged up by a tiny amount, roughly how many times that amount does the 49 move?

Look at each step on its own: doubling makes any nudge 2 times bigger.

Adding 1 passes a nudge through unchanged, so it multiplies it by 1.

Squaring near 7 makes a nudge about 14 times bigger, because 7 squared grows by about twice 7 for each small step.

The chain rule says the reactions of the steps multiply: 2 times 1 times 14.

Answer: About 28 times the nudge; starting from 3.01 gives about 49.28.

Practice

1. A photo app trains on batches of 32 pictures, and one layer takes 1,000 numbers per picture in and gives 1,000 numbers per picture out. Laid out in one long row as the book does, how many entries does the table connecting every number going in with every number coming out have?

Hint: First count all numbers going in for the whole batch, then all numbers coming out; the table has one entry for each pair.

Answer: 32 times 1,000 is 32,000 numbers in and 32,000 out, so the table has 32,000 times 32,000, about 1 billion entries. That is why one may know how to compute such a table and still not want to store it.

2. Explain in your own words why the book lets a whole layer, or even a whole smaller model, count as a single primitive.

Hint: Think of how programs are built from blocks that are themselves built from blocks.

Answer: The method only needs to know, for each step, how its output reacts to its input and to its parameters. Any block for which we know that can be treated as one step, whether it is a single multiplication, a layer or a whole model, just as programmers build big things from smaller blocks.

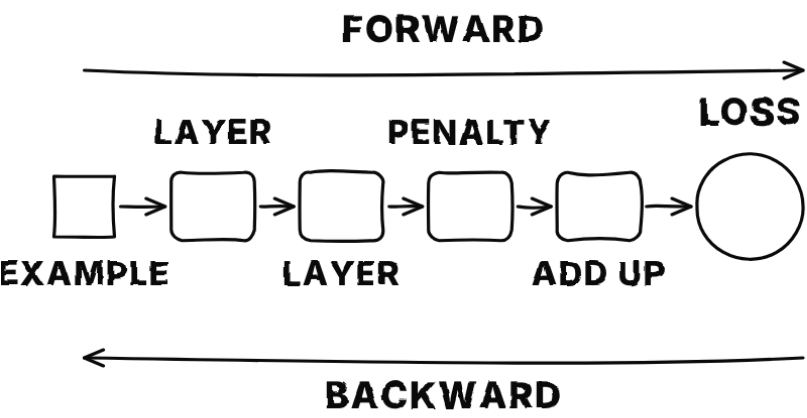
3. A friend says the ReLU cannot be used in this method because it has a sharp corner where its slope is undefined. What does this part say about such steps?

Hint: Separate steps with a single sharp corner from truly jumpy operations.

Answer: Steps like the ReLU, which are continuous but have a corner, can be brought into the method; the chapter promises to show how later. Truly jumpy operations, such as drawing a random sample, need other tricks and are left for the book's second volume.

Why the Obvious Ways Are Too Slow

Open the calculator on your phone and try the most direct way of finding out how a result reacts to a nudge. Work out the recipe from before, three doubled, plus one, squared, and note forty-nine. Then run it again from three point zero one and note the new result, a little over forty-nine point two. The difference, divided by the size of the nudge, tells you the reaction, about twenty-eight. It works, and it took two runs of the whole recipe to learn about one single starting number.



Before judging that method, the book pins down exactly what a training program looks like. Picture a row of stations on an assembly line. The raw example goes in at the first station, each station is one primitive with its own parameters, and each hands its result to the next. The early stations are the layers of the model; the last real station compares the model's answer for each example with the right one and gives each example a penalty, such as the cross-entropy you may meet when chat models are trained; and a final step simply adds up the penalties of the whole batch. So the line always ends in one single number, the loss, because one number is what an optimizer can push downhill. The book calls this written-out sequence of stations the evaluation trace of the program.

With that picture, the task gets a precise name. Automatic differentiation means computing, all at once and efficiently, how that final number reacts to the parameters of every station, using only what we know about each station by itself: its input Jacobian and its weight Jacobian. There are two great families of methods for doing it, which differ only in the order in which the stations' reactions are combined. One goes along the line in the direction it runs, forward mode; the other walks back from the end, backward mode, which neural network people call back-propagation. The book announces now, and shows soon, that backward mode is far cheaper for programs like ours. A real model is not always a straight line, and some models reuse the same parameters at several stations, as the recurrent models of a later chapter do; the same reasoning stretches to cover both.

Now back to the calculator. What you did is called numerical differentiation: nudge one number, run everything again, compare. For one number that is two runs; for a chat assistant with billions of

parameters it is billions of runs of the whole model for a single training step. That is hopeless as a training method, and the book keeps it for one modest job only, checking that a cleverer method gives the right answers.

$$\begin{array}{ccc}
 3 & \xrightarrow{\text{RECIPE}} & 49 \\
 3.01 & \xrightarrow{\quad\quad\quad} & 49.28
 \end{array}
 \left. \vphantom{\begin{array}{ccc} 3 & \xrightarrow{\text{RECIPE}} & 49 \\ 3.01 & \xrightarrow{\quad\quad\quad} & 49.28 \end{array}} \right] \text{DIFFER } 0.28$$

RUNS: 2 $0.28 \div 0.01 = 28$

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

REPEATED

Nudge and Rerun

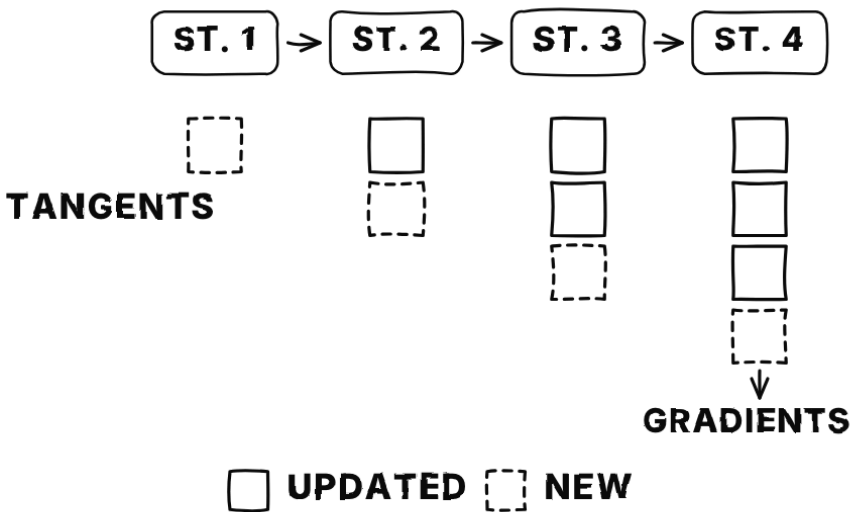
A second idea is to let a computer do school algebra. Hand a symbolic maths program a formula made of a sine and a few multiplications, and it returns the formula of its derivative, just as a student would on paper. The book shows a short Python listing that does exactly this with the SymPy library and prints the derivative as a new formula. The trouble is that a written-out formula repeats itself: the same sine appears in several places, and a sensible program would compute it once and keep the value. Finding the very best way to share all such

repeated pieces turns out to be one of those problems for which no fast general method is known. Automatic differentiation sidesteps that hunt by following the structure the program already has, station by station, and the book adds that it ends up equal to a symbolic method that is allowed to keep and reuse intermediate results.

Try it yourself

Write down the recipe start with three, double it, add one, square it, and run it twice on your phone's calculator, once from three and once from three point zero one. You should see the result move by about zero point two eight, twenty-eight times your nudge. Now imagine repeating those two runs for every one of a model's billions of numbers, and the need for something better becomes obvious.

So the book turns to the first genuine method, forward mode, and it rests on the chain rule. If one station feeds another, the reaction of the pair is the reaction of the second multiplied by the reaction of the first. Read it as bookkeeping along the line: once you know how the output of the first station reacts to its parameters, and the second station runs, you update that knowledge by multiplying it with the second station's input Jacobian.

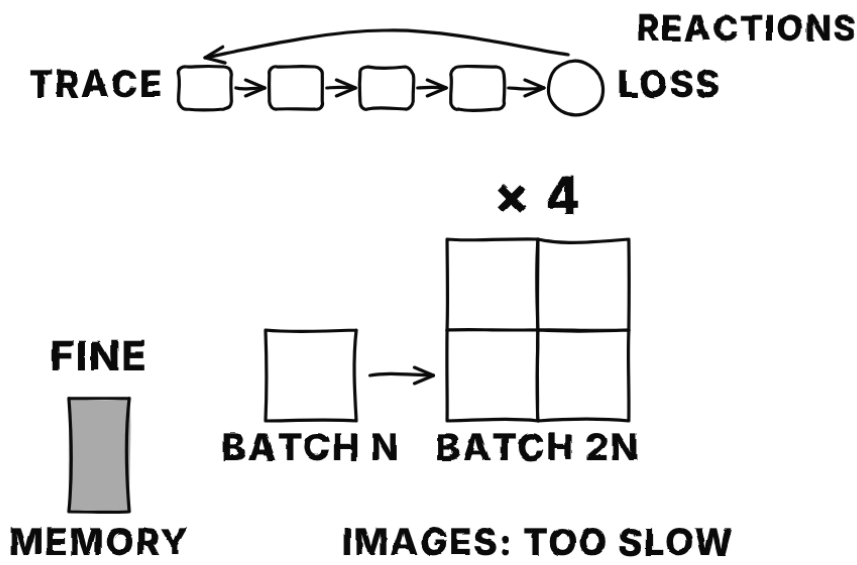


Forward Mode Keeps Tables

Forward mode keeps one running table for every set of parameters met so far, called a tangent. At the first station it starts a table for the first parameters, simply that station's weight Jacobian. At the second station it does two things: it updates the first table with the second station's input Jacobian, and it starts a fresh table for the second parameters. At every further station the same pattern repeats, so the tenth station must update nine stored tables and start a tenth, all using the same input Jacobian of that station. When the final adding-up step is reached, each table is collapsed into the gradient for its parameters, and the job is done.

The good news is memory: all this bookkeeping marches along with the program, so it needs roughly as much memory as the program

itself. The bad news is time. Each update multiplies two large tables of reactions, and when a layer handles a batch of examples, the cost grows with the square of the batch size and with the square of the number of values per example. Double the batch of photos a photo app trains on, and that part of the work grows fourfold; double the size of each picture's description, and it grows fourfold again. For high-dimensional inputs such as images, forward mode quickly becomes impossible, and that is the problem the backward walk will solve.



Memory Fine, Time Squared

Remember

A training program is a line of stations, the evaluation trace, that ends by adding up the penalties of a batch into one number, the loss. Automatic differentiation means finding how that one number reacts to the parameters of every station, efficiently and all at once, from each station's own reactions. Nudging each number and rerunning the model costs two runs per number, and writing out derivative formulas repeats work, so neither is usable for training. Forward mode follows the line, keeping and updating a table for every set of parameters met so far; it is thrifty with memory but each update multiplies huge tables. That cost grows with the square of the batch size and of the values per example, which rules forward mode out for large inputs such as images.

Questions

1. Why does a training program always end in one single number?
 - a. Because the model can only answer one example at a time
 - b. Because the penalties of the batch are added up, and one number is what an optimizer can push downhill
 - c. Because the first station of the line reduces every example to one number
 - d. Because the chain rule cannot handle more than one output

2. Why is nudging each number and rerunning the model useless for training a large chat assistant?
 - a. Because the answers it gives are always wrong
 - b. Because it only works for numbers smaller than one
 - c. Because it needs about two runs of the whole model for every single parameter, billions of runs per training step
 - d. Because it needs the derivative formula written out first
3. In forward mode, what must the tenth station of the line do?
 - a. Update the nine tables kept so far and start a tenth one
 - b. Wait until the final number is known and then walk back
 - c. Only start its own table and leave the earlier ones alone
 - d. Rerun the first nine stations with nudged numbers
4. If a photo app doubles the size of its training batch, what happens to the cost of forward mode's table updates?
 - a. It stays the same, because memory is what grows
 - b. It doubles, like the batch
 - c. It roughly quadruples, because it grows with the square of the batch size

Answers

1. **b)** The last real station gives each example a penalty and a final step adds them up; one number is what an optimizer can push downhill.
2. **c)** Numerical differentiation costs two runs per number, so a model with billions of parameters would need billions of runs per step; it is kept only for checking other methods.

3. a) At every station forward mode updates all stored tables with that station's input Jacobian and starts a fresh table for the station's own parameters.

4. c) Each update multiplies two large tables, and the cost grows with the square of the batch size and of the values per example.

Worked examples

Measuring a reaction by nudging The recipe is: start with 3, double it, add 1, square it. Run it from 3 and from 3.01 and use the two results to estimate how strongly the result reacts to the starting number.

From 3: doubled is 6, plus 1 is 7, squared is 49.

From 3.01: doubled is 6.02, plus 1 is 7.02, squared is 49.2804.

The result moved by 0.2804 while the start moved by 0.01.

Dividing the move of the result by the nudge gives the reaction: 0.2804 divided by 0.01.

Answer: About 28; it took two full runs to learn the reaction to one number.

Practice

1. A small model has 1,000 parameters. Using nudge-and-rerun in its simplest form, two runs per number, how many runs of the model does one training step need?

Hint: Each number to be differentiated needs its own pair of runs.

Answer: 2,000 runs for one step, instead of a handful; that is why the book keeps this method only for checking a cleverer one.

2. Explain in your own words why forward mode is thrifty with memory but too slow for photo models.

Hint: Ask what it has to store as it marches along, and what it has to multiply at each station.

Answer: Its bookkeeping marches along with the program, so it needs about as much memory as the program itself. But at every station it multiplies large tables of reactions for every set of parameters met so far, and that cost grows with the square of the batch size and of the values per example, which is impossible for images.

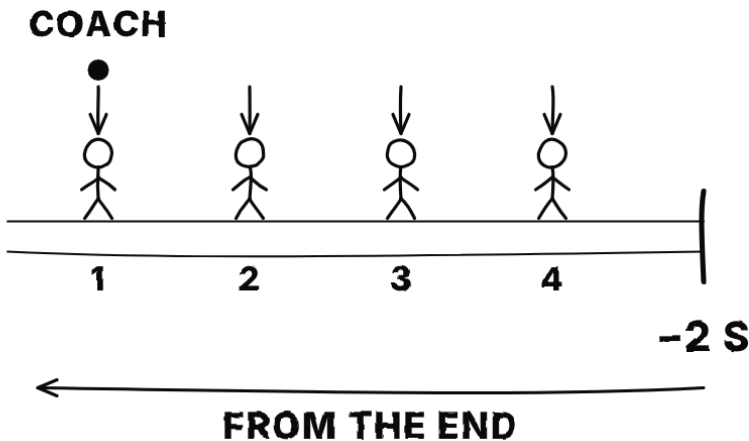
3. A symbolic maths program writes out the derivative of a formula in which the same sine appears several times. Why is that not yet a good way to differentiate a model?

Hint: Think of what a written-out formula does with pieces that repeat.

Answer: The written-out formula repeats the same pieces, while a sensible program would compute each once and reuse it; finding the best way to share all repeated pieces is a problem with no known fast general method. Automatic differentiation avoids the hunt by following the structure the program already has.

Walking Back From the Score

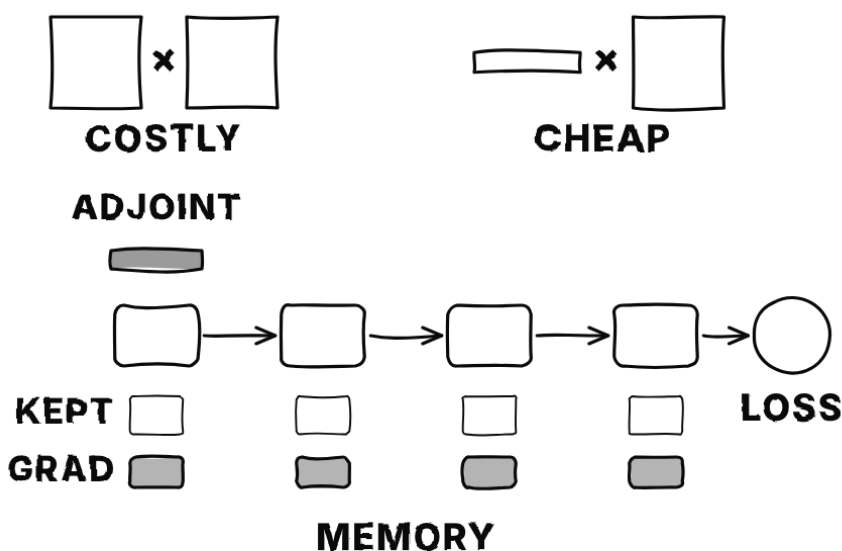
Think of a relay race that your team has just lost by two seconds. The coach does not ask every runner to rerun the course with slightly different shoes. She starts at the finish line, where there is one single number, the two seconds, and walks back along the track, asking each runner in turn how much of that number passed through their leg. Each answer is short, and each one prepares the next question for the runner before. That walk back from one final number is the whole idea of this part.



Walking Back From the Finish

Forward mode, as we saw, carried a big table for every set of parameters along the line and multiplied huge tables at every station. The book now looks at the chain of reactions for one set of parameters, written out in full. It is a long row of multiplications: at its far end stands the final adding-up step, then the input Jacobians of all the later stations, one after another, and finally the weight Jacobian of the station we care about. Multiplication of such tables can be done in any order and gives the same result. Forward mode simply worked through the row in the order the program ran; the trick is to start from the other end.

Starting from the end has two gifts. The first is that the end is tiny: because the program finishes in one single number, the last term is not a big table but a simple row of values, and multiplying a row by a table is far cheaper than multiplying two tables, and gives another row. The second is that the long stretch of input Jacobians, from the end back to any station, can be built up step by step, each time adding one more station on the left side of the walk. So one small row of values travels backwards, station by station, and at every stop it is reused rather than recomputed.

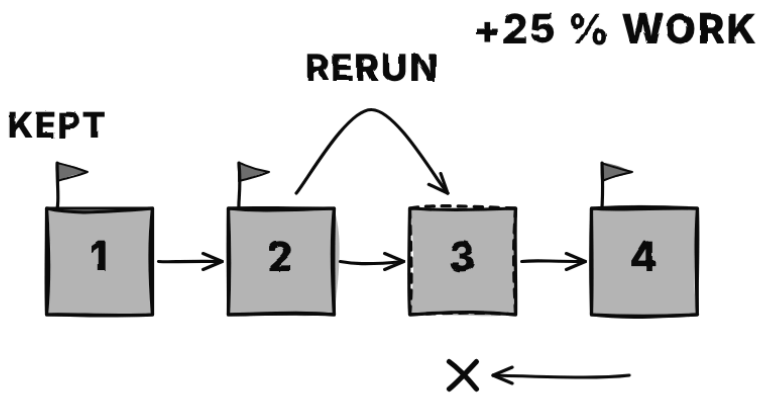


A Small Row Travels Back

Put together, this is reverse-mode automatic differentiation. First the whole program runs normally, and every intermediate result is kept. Then a row of ones is set up at the finish, standing for the final adding-up. Walking backwards, at each station two cheap things happen: the travelling row is combined with that station's weight Jacobian, which gives exactly the gradient for its parameters, and then it is combined with the station's input Jacobian, ready for the station before. The travelling rows are called adjoints; each one says how the final number reacts to the output of a station. The backward walk mirrors the original program, and practitioners call the two halves the forward pass and the backward pass, which is not to be confused with forward mode.

This is what happens every time a chat assistant learns from a batch: one forward pass to get the loss, one backward pass to get every gradient. Each step of the walk costs only in proportion to the sizes involved, never their squares, so reverse mode is dramatically faster than forward mode. The price is memory, because the backward walk needs the intermediate results of every station, and they must all be kept until it gets there.

The book shows a way to trade some of that memory back for time, called gradient checkpointing. Picture four blocks in a row. On the forward pass, only the outputs of the first, second and fourth blocks are kept; these are the checkpoints. The backward walk goes fine until it reaches the third block, whose results were thrown away, so it stops. The program then reruns the forward pass from the nearest checkpoint, the second block, to regenerate what was missing, and the walk continues. In this example the whole job costs a quarter more computation than an ordinary backward pass, and in general the fewer checkpoints you keep, the more recomputation you pay for.



Checkpoints Save Memory

Remember

The chain of reactions for any set of parameters is a long row of multiplications that may be done in any order. Because the program ends in one number, starting from the end means carrying a small row of values backwards instead of multiplying big tables, and each step reuses the previous one. Reverse mode runs the program once, keeps every intermediate result, then walks back, producing every gradient at modest cost; this is the forward pass and the backward pass of every trained model. Its weakness is memory, since all intermediate results must wait for the backward walk. Gradient checkpointing keeps only some results and recomputes the rest during the walk, trading extra computation for less memory.

Questions

1. Why is it cheaper to work through the chain of reactions starting from the final number?
 - a. Because the final number is small, so the tables near the end are empty
 - b. Because the program ends in one number, so the end is a small row, and a row times a table is far cheaper than a table times a table
 - c. Because the stations near the end have no parameters
 - d. Because walking backwards needs no intermediate results

2. What happens at each station during the backward walk?
 - a. The station is run again with nudged parameters
 - b. The travelling row is combined with the weight Jacobian to give that station's gradient, then with the input Jacobian for the station before
 - c. A new table is started for every earlier station
 - d. The station's intermediate result is thrown away and the walk continues
3. What is the main price of reverse mode compared with forward mode?
 - a. It needs much more time for each training step
 - b. It gives only approximate gradients
 - c. It needs a lot of memory, because every intermediate result must be kept until the backward walk reaches it
 - d. It can only be used on models with four blocks
4. If a training run keeps fewer checkpoints, what happens?
 - a. It uses less memory but has to recompute more during the backward pass
 - b. It uses more memory and computes less
 - c. Nothing changes, since checkpoints only affect the forward pass
 - d. The gradients become less accurate

Answers

1. b) The program finishes in one single number, so the last term is a simple row of values; multiplying a row by a table is far cheaper than multiplying two tables, and gives another row.

- 2. b)** At each station the travelling row gives the gradient for its parameters through the weight Jacobian and is then passed on through the input Jacobian, ready for the station before.
- 3. c)** Reverse mode is far faster, but the backward walk needs the intermediate results of every station, so all of them must be kept until it gets there.
- 4. a)** Checkpointing trades memory for time: the fewer results are kept, the more must be recomputed from the nearest checkpoint during the backward walk.

Worked examples

Checkpoints in four blocks A model has four blocks of equal cost. On the forward pass only the outputs of blocks one, two and four are kept. How much extra computation does the backward pass need, compared with keeping everything?

The backward walk passes block four and arrives at block three, whose results were not kept.

It reruns the forward pass from the nearest checkpoint, block two, to regenerate block three's results.

That rerun costs one block's worth of forward work, on top of the four blocks of the normal forward pass.

One extra block out of four is a quarter more computation.

Answer: About 1.25 times the computation, a quarter more, in exchange for not storing block three's results.

Practice

1. Explain in your own words why the order of the multiplications in the chain of reactions may be changed at all.

Hint: Think of what kind of operation the chain is made of.

Answer: The chain is just a long row of multiplications of tables, and such multiplications give the same result whichever pairs are combined first. So we are free to start from the final number instead of from the first station, and that choice is what makes reverse mode cheap.

2. A phone's photo model is too big to keep all intermediate results in memory during training. What can its engineers do, and what does it cost them?

Hint: Think of keeping only some results.

Answer: They can use gradient checkpointing: keep only a few intermediate results, the checkpoints, and recompute the others from the nearest checkpoint during the backward pass. The memory needed falls, and the price is extra computation, which grows the fewer checkpoints they keep.

3. Why must reverse mode run the whole program before it can compute any gradient, while forward mode could work along with the program?

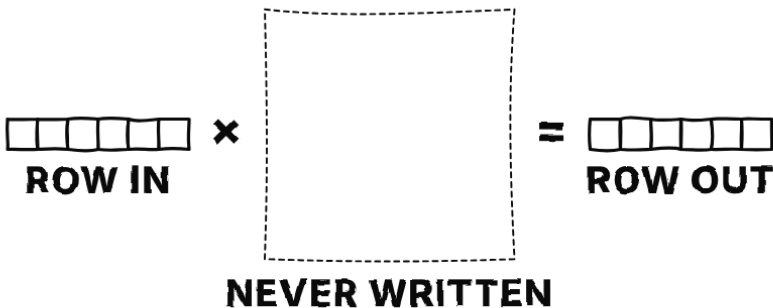
Hint: Where does the backward walk start?

Answer: The backward walk starts from the final number at the end of the program, so that number, and every intermediate result the walk will need, must exist first. Forward mode follows the program's own order, so its bookkeeping can march along with it.

Asking the Table One Question

Imagine a market stall with a huge price board: for every fruit, how much each kind of juice it goes into costs. You walk up holding a shopping list that says how many bottles of each juice you want. You do not need the stall owner to copy out the whole board for you; you need one answer, what your list costs per fruit. The owner can work that out in their head, row by row, without ever writing the board down. That difference, between handing over a whole table and answering one question put to it, is the trick of this part.

VJP



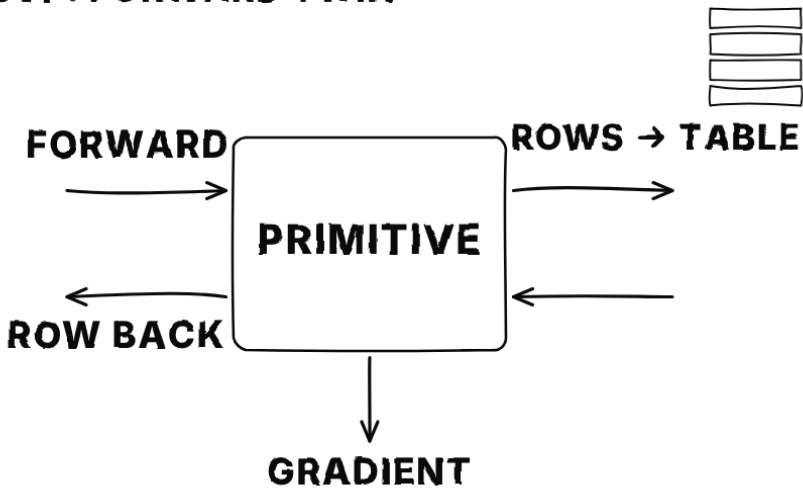
× 2 PER STATION

One Question, Not the Table

Look again at the backward walk. At every station only one kind of operation happened: the small travelling row was combined with one of the station's tables, either its input Jacobian or its weight Jacobian. The book gives that operation a name, the vector-Jacobian product, VJP for short: a row of "how much does each output matter" goes in, and a row of "how much does each input matter" comes out. Every station therefore needs exactly two of them, one for its input and one for its parameters, and one call of each is enough to carry the backward pass through that station.

Here is the consequence, and it is a big one. A whole automatic differentiation system can be built by choosing a set of primitives and giving each of them its two VJPs, without ever writing a full Jacobian into memory. The book draws it as a box for each primitive, with the usual forward door and two extra backward doors, one handing the travelling row back to the station before, the other handing out the gradient for the box's own parameters. Nothing is lost: if you ever wanted the full table, you could rebuild it one row at a time by asking the VJP about a single output at a time. For the mirror image, forward mode, the book notes that there is a twin operation going the other way, the Jacobian-vector product, JVP for short.

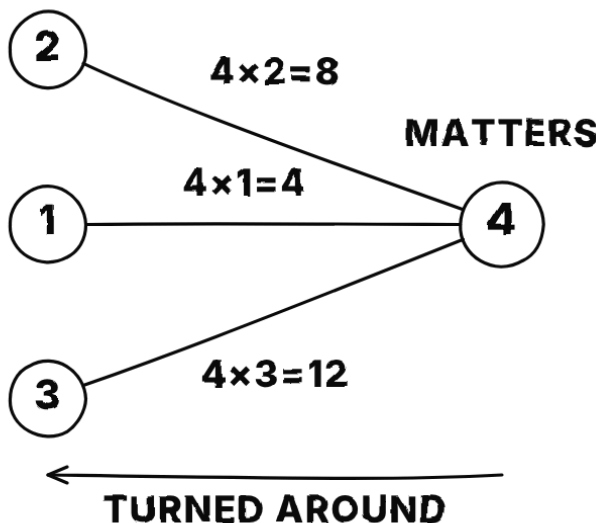
JVP: FORWARD TWIN



A Box With Two Back Doors

Why is this so much better? Take the most common station of all, a layer that mixes its inputs with a grid of weights, as every layer of a chat model or a photo model does many times over. Its table of reactions with respect to its weights is not even a flat table but a three-dimensional block, because every output can react to every weight in the grid. Yet the question the backward walk puts to it has a strikingly simple answer. To pass the travelling row back, the layer sends it through its own grid of weights, turned around so that outputs and inputs swap places. And the gradient for a single weight is just two numbers multiplied: how much the output it feeds matters, times the input value it multiplied on the way forward.

INPUTS NO 3-D BLOCK



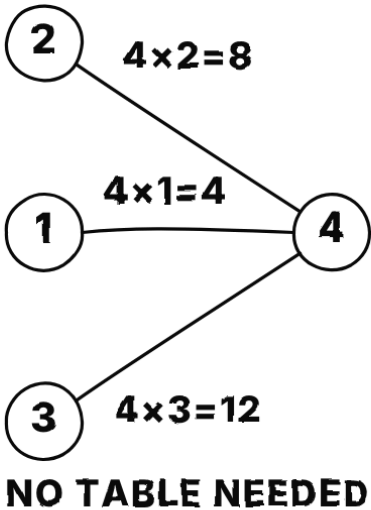
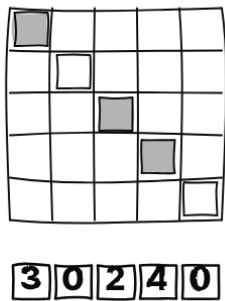
A Weight's Share of Blame

That last fact is worth saying again in other words. Each weight connects one input to one output, so its share of the blame is the importance of its output times the size of its input, and the whole grid of these products is filled in at once. The three-dimensional block never needs to exist.

The second common station is an activation function such as the ReLU, which treats every value on its own and has no parameters, so only one VJP is needed. Its table is almost entirely empty, because each output depends only on its own input; all that matters sits on the diagonal, the slope of the function at each value. So the backward question is answered by scaling each entry of the travelling row by that slope, one by one. For a ReLU, that means passing the value

through where the input was positive and stopping it where the input was negative. Again the full table, with its ocean of zeros, is never built.

RELU TABLE



ReLU Only Scales the Row

Try it yourself

Take three numbers as a layer's inputs, say two, one and three, and imagine one output that matters by an amount of four. The gradient for the weight linking that output to the first input is four times two, for the second four times one, for the third four times three. You have filled in a row of gradients with three multiplications and no table at all.

Remember

The backward walk only ever multiplies a small row by a station's table, and that operation is called a vector-Jacobian product. Each primitive needs two of them, one for its input and one for its parameters, and with those alone a complete automatic differentiation system can be built without storing any Jacobian. For a layer of weights, the row is passed back through the weights turned around, and each weight's gradient is the importance of its output times the input it multiplied. For an activation such as the ReLU, the row is simply scaled, value by value, by the function's slope. In both cases the huge tables of reactions are never written down, which is what makes training large models possible.

Questions

1. Why can an automatic differentiation system work without ever storing a full Jacobian?
 - a. Because Jacobians are always small enough to be ignored
 - b. Because the backward walk only ever needs a row times a table, and each primitive can answer that directly through its two VJPs
 - c. Because the full Jacobian is computed once and then thrown away
 - d. Because only forward mode ever needs Jacobians

2. In a layer that mixes its inputs with a grid of weights, what is the gradient for a single weight?
 - a. The importance of the output it feeds times the input value it multiplied
 - b. The weight's own value times the size of the grid
 - c. The sum of all inputs of the layer
 - d. The slope of the activation function at that weight
3. Why does a ReLU station need only one VJP instead of two?
 - a. Because it has no parameters, so there is no gradient for weights to compute
 - b. Because its slope is always one
 - c. Because it is always the last station of the model
 - d. Because the forward mode handles it instead
4. If you really wanted the full table of a station, what could you do?
 - a. Nothing; once VJPs are used the table is lost forever
 - b. Rebuild it one row at a time by asking the VJP about a single output at a time
 - c. Run the forward pass twice and compare
 - d. Multiply all stations' VJPs together

Answers

1. b) The only operation the backward walk needs is the vector-Jacobian product, so giving each primitive its two VJPs is enough, and no Jacobian is ever written into memory.

- 2. **a)** Each weight connects one input to one output, so its share of the blame is the importance of its output times the size of its input.
- 3. **a)** An activation such as the ReLU treats every value on its own and has no parameters, so only the VJP for its input is needed.
- 4. **b)** Nothing is lost: asking the VJP about a single output at a time gives the full table one row at a time.

Worked examples

A grid of weight gradients A small layer takes three inputs, 2, 1 and 3, and has two outputs. The travelling row says the first output matters by 4 and the second by 1. What are the gradients of the six weights?

Each weight links one input to one output, so its gradient is that output's importance times that input.

For the first output, which matters by 4: 4 times 2, 4 times 1, 4 times 3, giving 8, 4 and 12.

For the second output, which matters by 1: 1 times 2, 1 times 1, 1 times 3, giving 2, 1 and 3.

Together these fill the whole grid of weight gradients without building any three-dimensional block.

Answer: First output's weights: 8, 4, 12; second output's weights: 2, 1, 3.

Practice

1. A ReLU received the inputs 1.5, minus 0.7 and 2 on the forward pass. On the backward walk the travelling row arriving at it is 5, minus 2 and 3. What row does it pass back?

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

Answer: 5, 0 and 3: the first and third values pass through because their inputs were positive, and the second is stopped because its input was negative.

2. Explain in your own words why a weight's gradient should be the importance of its output times the input it multiplied.

Hint: Think of what one weight does on the forward pass.

Answer: On the forward pass a weight multiplies one input and adds the result into one output. Turning the weight up therefore moves that output in proportion to the input, and the output's effect on the final number is its importance; so the weight's effect is the two multiplied together.

3. Why is the table of a ReLU described as an ocean of zeros, and why does that matter?

Hint: On what does each output of the ReLU depend?

Answer: Each output depends only on its own input, so the table has values only on its diagonal and zeros everywhere else. Building it would waste memory on zeros, while the VJP simply scales each value by the slope, which needs no table at all.

How Frameworks Walk Back, and Why ReLU Won

Put a page on a photocopier, set it to reduce to a quarter of its size, and copy the copy, and copy that copy again. After two rounds the text is a smudge; after ten, the page is a speck you could cover with a grain of sand. Each round only shrank things by a fixed factor, but the factors multiplied. Keep that photocopier in mind; it is about to explain why one small function took over modern AI. First, though, the book looks at how real frameworks carry out the backward walk.



PYTORCH: TORCH.FUNC

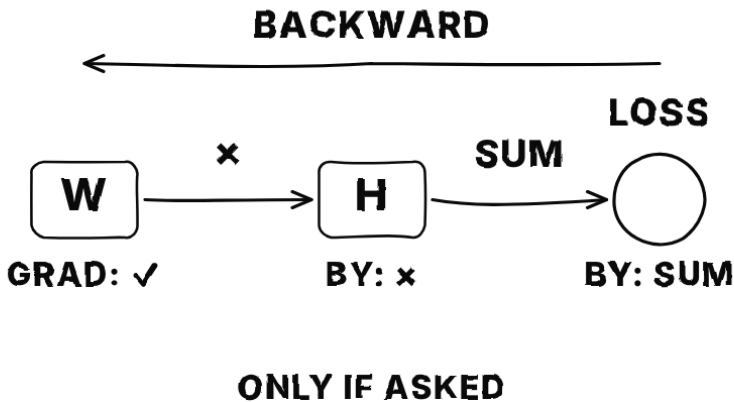


A Tool That Returns a Function

There are many ways to build reverse mode in software, from keeping a running list of every operation performed, as TensorFlow does, to rewriting the program's code into its backward twin. In JAX, where everything is a function, the tools themselves are functions that take functions. You hand JAX a function and get back a new function that computes its gradient. In the same spirit, the VJP tool takes a function and a point and hands back a little function that answers the backward question at that point. The book's short example defines a function that squares ten numbers and adds them up, asks for its gradient function, and feeds it ten random numbers; out come ten numbers, one reaction for each input. PyTorch offers the same style in a module of its own, and by combining these tools cleverly one can get even more, such as second derivatives.

Most models, though, are written as objects, layers that keep their parameters inside them, as we saw at the start. One way to bring them into the function style is to "purify" them: pull the parameters out into a separate collection and call the model as a plain function of its input and those parameters. PyTorch also has its own built-in way, which needs no such step. Every block of numbers it computes carries two extra notes: an empty slot for its future gradient, and a record of the operation that produced it. The book's figure shows it: on the left, the numbers with their two notes; on the right, the backward walk following the records from the final loss back through the graph, dropping gradients into the slots on its way. To save memory, slots are filled only for the numbers that have asked for a gradient, typically the parameters being trained. That is literally what happens inside the training of a photo app's model when the program calls for the backward step on the loss, and the author warmly recommends

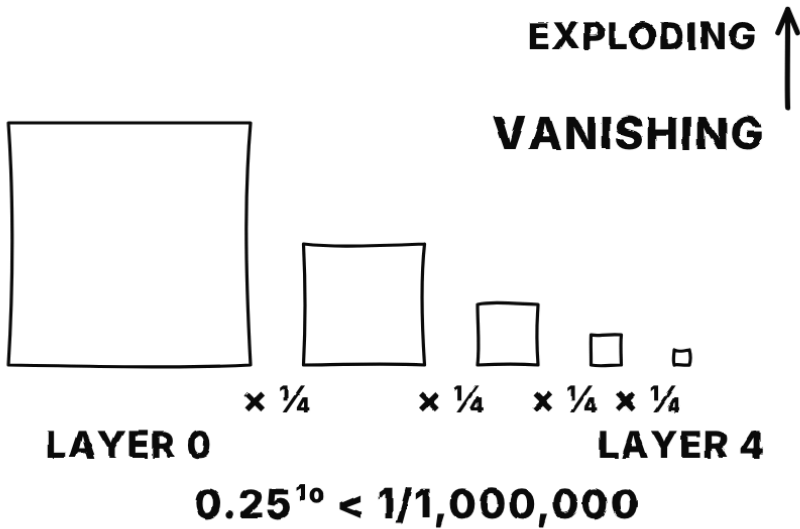
building a tiny version of such a system yourself, as many teaching projects online do.



Tags Followed Backwards

Now back to the photocopier. Remember that when the backward walk passes an activation function, the travelling row is scaled by the function's slope. A deep model has many such stations, so the row is scaled again and again, and the factors multiply exactly like the copier's reductions. If the slopes are below one everywhere, the gradient shrinks towards nothing at a rate that grows with every layer, which is called the vanishing gradient problem. If they are above one everywhere, it grows without bound, the exploding gradient problem. Both are real dangers because computers store numbers with limited

precision, typically thirty-two binary digits or fewer, so a number that gets too small becomes zero and one that gets too large overflows.

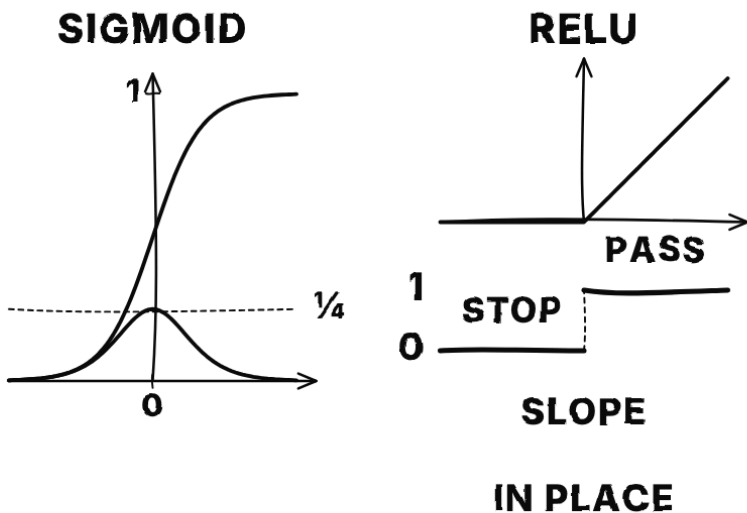


Factors That Multiply

Try it yourself

On your phone's calculator, type zero point two five and multiply it by zero point two five ten times. You should end with less than one millionth. That is roughly what ten layers of the old favourite activation can do to a gradient at best, and real networks have far more layers.

That old favourite is the sigmoid, a smooth S-shaped curve that was popular because it is a soft version of an on-off switch. Its slope is a low hump that never rises above a quarter, highest in the middle of the S and almost flat at both ends. So every sigmoid layer shrinks the backward row to a quarter or less, the photocopier set to its smallest setting.



Sigmoid Slope vs ReLU Slope

The ideal would be a function whose slope is one everywhere, but the only such function is the one that changes nothing, and stacking layers that change nothing would make the whole model no more powerful than a single layer. So we need a function straight enough to keep gradients healthy, yet bent enough to keep the layers apart. The ReLU fits: it is flat at zero for negative inputs and a straight line for

positive ones, so its slope is either zero or exactly one. On the backward walk each value is therefore either stopped, which makes the computation sparse, or passed on untouched, with no shrinking and no blowing up. The book's figure shows the hinge-shaped ReLU beside its slope, a step that jumps from zero to one.

In a side box, the author adds a curiosity: a stack of purely linear layers, run on a real computer, is not exactly linear, because of tiny roughness at the limit of the machine's precision, and this has even been exploited to train fully linear deep networks.

The ReLU has one more, small gift. Its slope can be read from its output just as well as from its input, because it only blanks out negative values and leaves positive ones as they are. So a framework may overwrite the ReLU's input with its output during the forward pass and still get the backward walk right, saving a little of that precious memory; PyTorch has an in-place option for exactly this.

Remember

JAX treats gradients and VJPs as tools that take a function and return a new function, while PyTorch tags every computed block of numbers with the operation that made it and a slot for its gradient, then walks those records back from the loss. Each activation scales the backward row by its slope, and over many layers those factors multiply. Slopes below one make gradients vanish and slopes above one make them explode, which limited computer precision turns into real failures. The sigmoid's slope is at most a quarter, so deep stacks of it starve the gradient; the ReLU's slope is either zero or one, so values are either stopped or passed on untouched. Because the ReLU's slope can be read from its output, its input can be overwritten in place to save memory.

Questions

1. When PyTorch runs the backward step on the loss, how does it find its way back through the model?
 - a. It reruns every layer with nudged numbers
 - b. It follows the record each block of numbers keeps of the operation that produced it, dropping gradients into slots on the way
 - c. It asks JAX to compute the gradient function
 - d. It writes out the derivative formula of the whole model first

2. Why do deep stacks of sigmoid layers tend to starve the gradient?
 - a. Because the sigmoid has no slope at all
 - b. Because its slope is never above a quarter, and the slopes of all layers multiply on the backward walk
 - c. Because the sigmoid overwrites its input in place
 - d. Because sigmoid layers have too many parameters
3. Why not simply use an activation whose slope is one everywhere?
 - a. Because such a function changes nothing, and stacked layers would be no more powerful than one
 - b. Because a slope of one makes gradients explode
 - c. Because computers cannot store the number one
 - d. Because it would need a gradient slot for every value
4. Why may a framework overwrite a ReLU's input with its output during the forward pass?
 - a. Because the ReLU's slope can be read from its output just as well as from its input
 - b. Because the ReLU never appears in the backward walk
 - c. Because the input is no longer needed by any later layer
 - d. Because overwriting makes the gradient larger

Answers

1. **b)** Every block of numbers carries a record of the operation that made it and a slot for its gradient; the backward walk follows those records back from the loss and fills the slots.

2. b) Each activation scales the backward row by its slope; the sigmoid's slope is at most a quarter, so over many layers the gradient shrinks towards nothing.

3. a) The only function with slope one everywhere is the one that changes nothing; the activation must be bent enough to keep the layers apart.

4. a) The ReLU only blanks out negative values and leaves positive ones unchanged, so its slope can be read from the output, and the input can be overwritten to save memory.

Worked examples

The gradient of a sum of squares As in the book's example, a function squares its inputs and adds them up. For the three inputs 1, minus 2 and 3, what does its gradient function return?

The gradient has one entry per input: how much the sum reacts to a nudge of that input.

Each input appears only in its own square, so only that square matters for it.

A square reacts to a small nudge at twice the value being squared.

So each entry is twice the input: 2 times 1, 2 times minus 2, 2 times 3.

Answer: 2, minus 4 and 6: as many numbers as there were inputs, just as the book's ten inputs gave ten numbers.

Practice

1. A model has five sigmoid layers, and at best each scales the backward row by a quarter. By how much, at most, is the gradient reduced after passing all five?

Hint: The factors multiply: a quarter times a quarter, five times over.

Answer: A quarter to the fifth power is 1 in 1,024, so the gradient is at best reduced about a thousandfold.

2. Now replace the five sigmoids by five ReLUs, and suppose the value in question is positive at every layer. What happens to it on the backward walk?

Hint: What is the ReLU's slope for positive inputs?

Answer: The slope is exactly one at every layer, so the value is passed back untouched, with no shrinking and no blowing up. Had it been negative at some layer, it would have been stopped there.

3. Explain in your own words why vanishing and exploding gradients are a real problem on a computer and not just on paper.

Hint: Think of how many digits a computer keeps for each number.

Answer: Computers store numbers with limited precision, typically 32 binary digits or fewer. A gradient multiplied by many small slopes eventually becomes too small to store and turns into zero, while one multiplied by many large slopes becomes too large and overflows, so training breaks down.

The Corner of the ReLU

Fold a postcard into a shallow V and stand it on the table, point down, like the graph of a ReLU seen with its corner at the bottom. Now take a ruler and try to rest it against the corner from below, so that it touches the corner and stays under the whole card. On a smooth curve there would be exactly one way to do it, the tangent. At the corner, the ruler can tilt anywhere between the slopes of the two sides, and every one of those tilts works. That small observation is how the book settles the last loose end of the chapter.

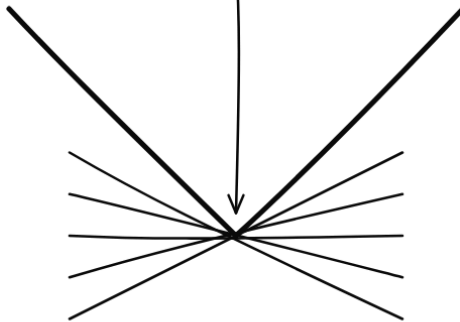
SMOOTH



RELU AT 0



CARD



CORNER: MANY

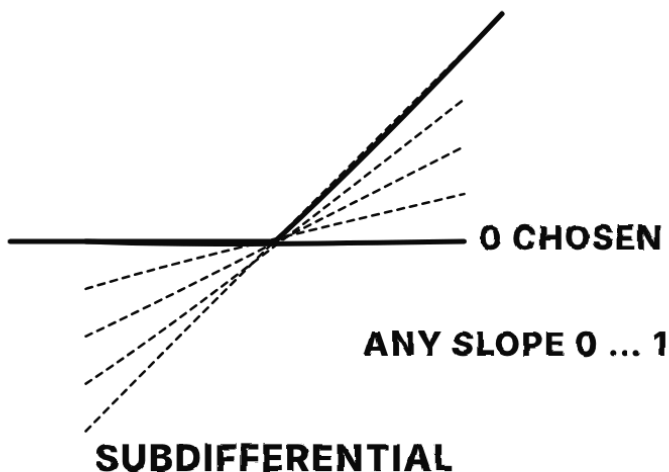
A Ruler Under a Folded Card

The ReLU, flat for negatives and a straight line for positives, has a corner at zero where its slope is not defined. What does the backward walk do there? The practical answer is that it hardly matters: training starts from random numbers and moves in small random steps, so the chance of landing exactly on zero is practically nil, and everywhere else the slope is perfectly clear.

The careful answer uses the idea of the ruler. For a function that curves upward like a bowl, any line that touches it at a point and stays below the whole function is a subgradient there, and its slope is a valid substitute for the missing slope. Where the function is smooth there is only one such line, the ordinary tangent; at a corner there are many, and together they form the subdifferential. For the ReLU at zero, every slope from zero to one qualifies, and most frameworks simply choose zero. Training that picks such a slope at every step is called subgradient descent.

RELU $\rightarrow 0$

RELU₂ $\rightarrow 1$ RELU₃ $\rightarrow 0.5$

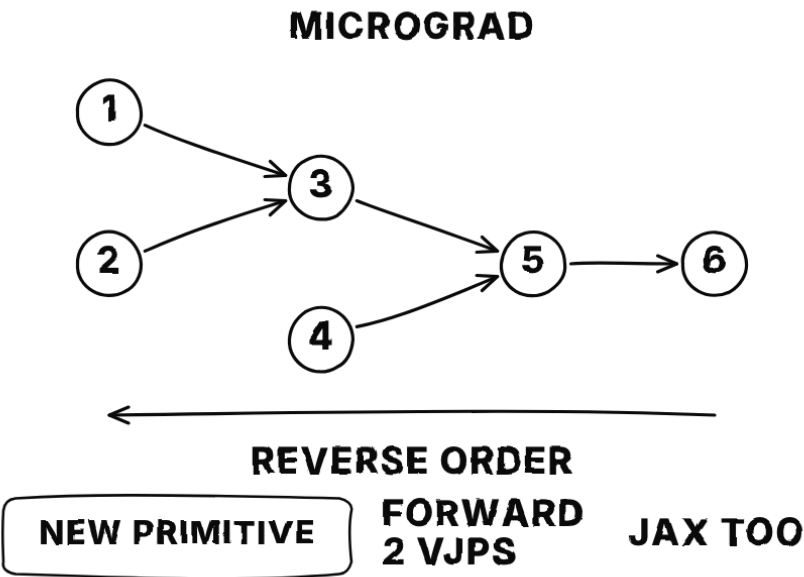


Every Slope From 0 to 1

The book admits that reality is messier. For functions that are not bowl-shaped, the definition has to be loosened to hold only near the point, and AD systems can give different answers depending on how a function is written. The author's example is striking: three ways of writing what is the very same ReLU make PyTorch report, at zero, a slope of zero, of one, and of one half.

To close, the book turns to practice. Its warmest advice is to build a small reverse-mode system yourself, as the teaching project micrograd does for single numbers. The one extra care needed when a model branches is to put the stations in a proper order before the backward walk, so that no gradient travels down a wrong path; micrograd does this with a cheap sorting step. A second good exercise

is to write a new primitive in PyTorch, giving it a forward pass and its VJPs, such as one of the trainable activations of the previous chapter, even though PyTorch could work out the backward pass by itself. The same can be done in JAX, which offers a teaching version of itself, a way to write custom VJPs, and a cookbook of advanced uses such as second derivatives. And with that, the backward walk that trains every chat assistant and photo app you use is no longer a mystery but a method you could write yourself.



Build Your Own Backward Walk

Remember

The ReLU has a corner at zero, but training almost never lands exactly there, so in practice its slope is always defined. Formally, a subgradient is the slope of a line that touches a bowl-shaped function at a point and stays below it everywhere; at a corner many such slopes exist, forming the subdifferential. For the ReLU at zero any slope from zero to one is valid, and frameworks usually take zero. Different ways of writing the same function can give different slopes at a corner, as PyTorch's zero, one and one half show. The best way to make the chapter your own is to build a tiny reverse-mode system or a custom primitive with its VJPs.

Questions

1. Why does the ReLU's undefined slope at zero hardly matter in everyday training?
 - a. Because frameworks remove every zero from the data
 - b. Because training starts from random numbers and moves in small random steps, so landing exactly on zero is practically impossible
 - c. Because the ReLU's slope at zero is always one
 - d. Because the backward walk skips all ReLU stations
2. At a corner of a bowl-shaped function, why are there many subgradients instead of one?

- a. Because many lines can touch the corner while staying below the whole function
 - b. Because the function has many corners next to each other
 - c. Because the function is not bowl-shaped at a corner
 - d. Because every line through the corner is a subgradient, whatever its tilt
3. Which slopes are valid subgradients of the ReLU at zero?
- a. Only zero
 - b. Only one
 - c. Any slope from zero to one
 - d. Any slope at all
4. What does the book's PyTorch example show about corners?
- a. That PyTorch refuses to compute gradients at corners
 - b. That three ways of writing the same ReLU give three different slopes at zero
 - c. That the ReLU is not really equal to its rewritten versions
 - d. That only JAX handles corners correctly

Answers

- 1. b)** The chance of landing exactly on zero is practically nil, and everywhere else the ReLU's slope is perfectly clear.
- 2. a)** A subgradient is the slope of a line touching the function at the point and staying below it; at a smooth point only the tangent does this, at a corner a whole range of tilts does.
- 3. c)** For the ReLU at zero every slope from zero to one qualifies, and most frameworks simply choose zero.

4. b) Three ways of writing the very same ReLU make PyTorch report at zero a slope of zero, of one and of one half.

Worked examples

Three ReLUs, three slopes A second version of the ReLU is written as: the ReLU of minus the input, plus the input. A third takes half of the original ReLU plus half of the second. Check that both equal the ReLU, and find the slope PyTorch reports at zero for each, given that it takes the ReLU's own slope at zero as 0.

For a positive input, the ReLU of minus the input is zero, so the second version gives the input itself, like the ReLU; for a negative input, it gives minus the input plus the input, which is zero, again like the ReLU.

The third version averages two functions that both equal the ReLU, so it equals the ReLU too.

At zero, the second version's slope is taken as minus the ReLU's slope, which is 0, plus 1 from the added input, so 1.

The third version's slope is half of 0 plus half of 1, so 0.5.

Answer: All three are the same function, yet PyTorch reports slopes of 0, 1 and 0.5 at zero.

Practice

1. A line passes through the ReLU's corner at zero with slope one half. Is its slope a valid subgradient there? What about a line with slope one and a half?

Hint: Test each line at an input of 2 and of minus 2: it must stay below or touch the ReLU everywhere.

Answer: With slope one half the line is at 1 when the ReLU is at 2, and at minus 1 when the ReLU is at 0, so it stays below: valid. With slope one

and a half it is at 3 when the ReLU is at 2, above the function, so it is not a subgradient.

2. Why must the stations be put in a proper order before the backward walk once a model branches?

Hint: Think of a station whose output feeds two later stations.

Answer: A station can pass its gradient on only after everything that depends on it has handed its share back. Without a proper order the walk could pass on an incomplete gradient and follow a wrong path; micrograd prevents this with a cheap sorting step.

3. Describe in your own words what you must give a new primitive so that a framework can train through it.

Hint: Recall the two backward doors of each primitive.

Answer: Its forward pass, which computes the output, and its VJPs: one that passes the travelling row back to the station before, and one that gives the gradient of its own parameters, if it has any.