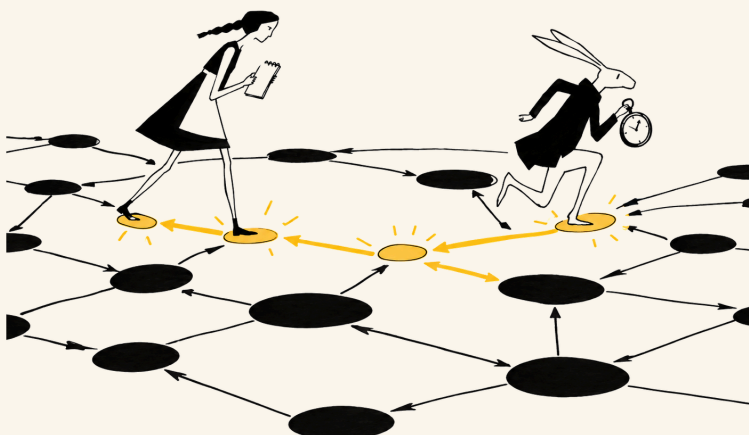


Alice's Adventures in a Differentiable Wonderland



Simone Scardapane · 2024

Illustrated audio lectures

Lecturized

Alice's Adventures in a Differentiable Wonderland

Illustrated audio lectures

Simone Scardapane · 2024

Retold as illustrated lectures by Lecturized
lecturized.com

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

Contents

A Program That Knows Its Own Slopes

One Number at the End, and the Slow Way Forward

Walking the Chain Backwards

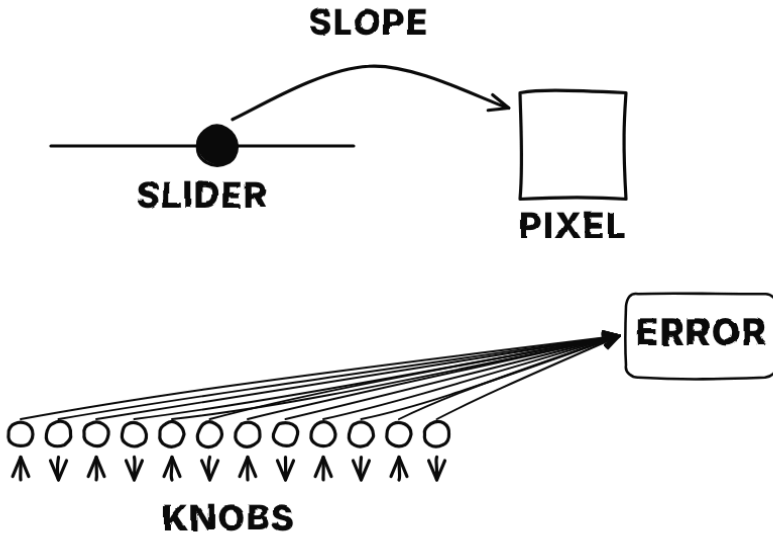
Every Piece Learns to Answer Backwards

How Libraries Do It, and Why the ReLU Won

The Corner of the Ramp

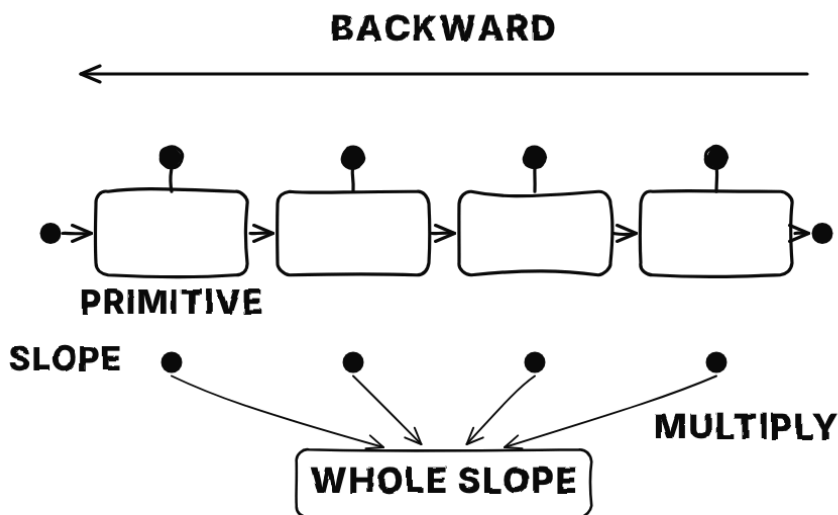
A Program That Knows Its Own Slopes

Take a photo with your phone and open the editing app. Drag the brightness slider a little to the right, and watch one pixel of the sky: it gets a touch lighter, and you can feel how much each nudge of the slider changes it. That feeling, "if I turn this knob a little, the result moves by so much", is what mathematicians call a derivative. Now imagine not one slider but a hundred million of them, all inside the model that sorts your photos or answers your chat questions, and a single number at the end that says how wrong the model was on a batch of examples. Training means turning every one of those knobs a little in the right direction, again and again, so the question this chapter answers is: how does a computer find out, quickly, which way to turn each knob?



Many Knobs, One Number

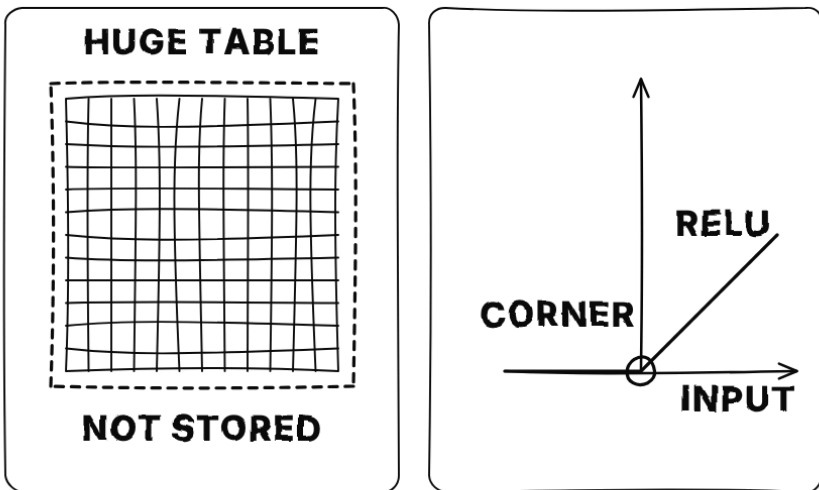
The book's answer starts by looking at the model as a program. When the program runs, it performs a chain of small, simple operations, one feeding the next; the book calls each one a primitive, and the whole chain, as it actually ran, a computational graph. A primitive can be as small as one multiplication of a table of numbers, as large as a whole layer, or even a whole model, because pieces built from pieces are still pieces. Each primitive takes something in, holds some adjustable numbers of its own, the weights, and passes something out.



A Chain of Primitives

Everything rests on one modest assumption. For every single primitive, we already know how its output responds to small changes, both in what came in and in its own weights; the book calls these two tables of slopes the input Jacobian and the weight Jacobian. Knowing the slopes of every piece, the rule of school calculus for a function inside a function, the chain rule, tells us in principle the slope of the whole program: just multiply the pieces' slopes together. So the problem looks solved before it starts, and the whole difficulty is doing that multiplication cleverly enough to be fast. The clever way is called back-propagation, or reverse-mode automatic differentiation, and it is the engine under every system that trains your chat assistant.

Two warnings come with the setup. The tables of slopes can be enormous, since every input number can affect every output number and a whole batch of photos is processed at once; a good method must use them without ever writing them out in full. And some pieces have corners, like the ReLU, which simply cuts negative numbers to zero; the book promises a way to handle such corners later in the chapter.



Two Warnings

Finally, the book notes that the way you write the program shapes how the slopes are found. In a library like JAX, everything is a function, and the weights are handed in like any other input, exactly as in the book's notation. In PyTorch or TensorFlow, a layer is an object that keeps its weights inside itself, which is convenient, for example to create the weights only once the size of the input is known,

but it hides the weights one step away from the mathematics. Left out here are the shapes of the slope tables for batches of data and how the book flattens them into plain lists; you will find them in the subsection on notation.

Remember

Training a model means finding, for every adjustable number, which way to turn it to make one final error smaller. The book sees the model as a running program, a chain of simple primitives, and assumes the slopes of each piece are known. The chain rule then gives the slope of the whole, so the real challenge is computing it fast and without storing gigantic tables.

Function-style libraries like JAX and object-style ones like PyTorch represent the weights differently, and that shapes how automatic differentiation is built in each.

Questions

1. If the slopes of every single primitive are already known, why is finding the slope of the whole program still a real problem?
 - a. Because the chain rule only works for programs with one primitive
 - b. Because multiplying all those slopes together has to be done cleverly to be fast
 - c. Because the slopes of the primitives change every time the program runs

- d. Because the final error number cannot be computed exactly
2. Why must a good method use the tables of slopes without ever writing them out in full?
- a. Because the tables can be enormous: every input number can affect every output number, for a whole batch at once
 - b. Because the tables are secret parts of the model that the library hides
 - c. Because writing them out would change the weights of the model
 - d. Because the tables only exist for primitives without corners
3. What does training the model that sorts your photos actually need from automatic differentiation?
- a. The final error number, computed as quickly as possible
 - b. A list of all primitives in the order they ran
 - c. For every adjustable number, which way to turn it to make the final error smaller
 - d. The exact shape of every table of slopes
4. How does PyTorch differ from JAX in the way it represents a layer's weights?
- a. PyTorch hands the weights in like any other input, JAX keeps them inside an object
 - b. PyTorch has no weights, only inputs and outputs
 - c. Both treat weights exactly as in the book's notation
 - d. PyTorch keeps the weights inside the layer object, JAX hands them in like any other input

Answers

1. **b)** The chain rule says the slope of the whole is the product of the pieces' slopes, so the problem looks solved; the whole difficulty is doing that multiplication cleverly enough to be fast.
2. **a)** Every input number can affect every output number, and a whole batch of photos is processed at once, so the tables of slopes grow far too large to store.
3. **c)** Training means turning every one of the many knobs a little in the right direction, so the computer must find out, quickly, which way to turn each one.
4. **d)** In JAX everything is a function and the weights are handed in like any other input, while in PyTorch a layer is an object that keeps its weights inside itself.

Practice

1. Think of the photo editing app with its brightness slider. In your own words, explain what a derivative tells you, and why a model with a hundred million such sliders needs a fast way to find all of them.

Hint: Start from what happens to one pixel when you nudge one slider a little.

Answer: A derivative says how much the result moves when you turn one knob a little. Training must turn every knob of the model in the right direction again and again, so it needs all these slopes, for all hundred million knobs, each time; doing that slowly would make training impossible.

2. A primitive can be one multiplication, a whole layer, or a whole model. Explain why the method of this chapter still works when the pieces are this different in size.

Hint: Ask what the method needs to know about each piece, whatever its size.

Answer: The method only assumes that, for each piece, we know how its output responds to small changes in what came in and in its own weights. Pieces built from pieces are still pieces, so a layer or a whole model can be treated just like a single operation.

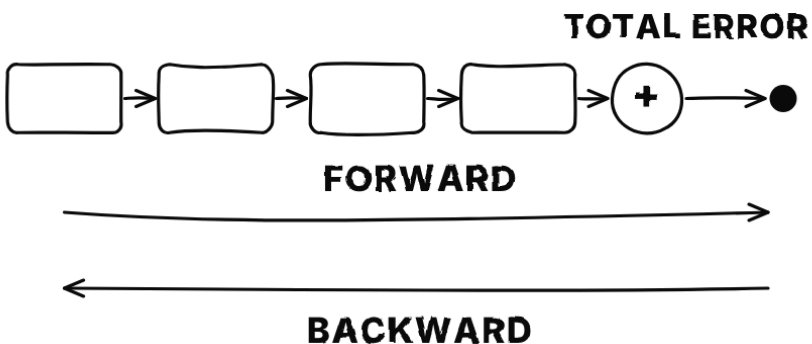
3. Some pieces, like the ReLU, have a corner where they cut negative numbers to zero. Why could such a corner be a worry for a method built on slopes?

Hint: Picture the graph of the ReLU right at the point where it bends.

Answer: At a corner the graph has no single slope, so the assumption that every piece's slope is known breaks there. The book promises a way to handle such corners later in the chapter.

One Number at the End, and the Slow Way Forward

Picture the simplest way anyone would find out which way to turn a knob. You run the model on a batch of photos and write down how wrong it was; you nudge one knob a hair; you run the whole model again and compare. The difference tells you the slope for that one knob, and it really works. Then you remember that the model behind your photo app has many millions of knobs, and each one asks for its own pair of runs. A single step of training would need millions of complete runs, which is why nobody trains a model this way.



Nudge, Rerun, One Number

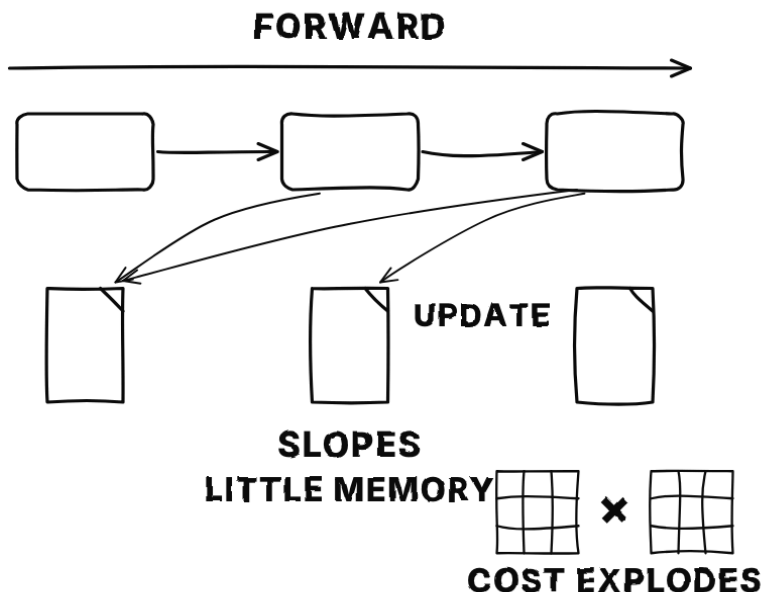
The book now sets the task down precisely. A model at work is a chain of primitives run one after another, each passing its result to the next, with the last link measuring the error on each example and a final step adding those errors up for the whole batch. The book calls this written-out run the evaluation trace of the program. What matters is that the chain always ends in one single number, the total error, because one number is what an optimiser can make smaller. Automatic differentiation is then the job of finding, all at once and efficiently, the slope of that one number with respect to every set of weights in the chain, using only the chain itself and the known slopes of each piece.

The book adds that there are two families of methods for this job, forward mode and backward mode, and that they differ only in the order in which the pieces' slopes are combined. It announces already here that the backward one, the back-propagation of neural networks, is far better for our purpose. The same ideas carry over to chains that branch and merge, and to models that use the same weights in several places, which returns in the chapter on recurrent models.

Before taking either road, the book sets aside two other ways of differentiating. The first is the experiment you just pictured, nudging and re-running, called numerical differentiation; at two runs for every single number, it is useful only as a spot check that a cleverer method gives the right answer. The second is symbolic differentiation, the way a computer algebra program does school calculus: you hand it a formula, and it hands back the formula of the derivative. The book shows a tiny example in a few lines of Python using the SymPy library, a short formula built from a sine, whose derivative the program prints out.

The trouble with the symbolic way is waste. The derivative formula repeats pieces of the original, the same sine and the same cosine, and a sensible program would compute each of them once and keep it. Finding the very best way to share such work, so that nothing is ever computed twice, turns out to be one of those problems for which no fast general method is known. Automatic differentiation sidesteps this by using what we already have, the chain of steps the program actually ran, and so does far better than the plain symbolic approach.

Now the first road, forward mode. Think of it as walking along the chain in the direction the program runs, carrying a notebook. Each time a new piece runs, you open a fresh page for its weights, recording how its output depends on them, and you also go back and update every page you have written so far, because the new piece passes all earlier influences on. When you reach the end, adding up the final errors turns each page into the slope you wanted. The good side is that this bookkeeping runs right alongside the program, so it needs about as much memory as the program itself.



Forward Mode: A Notebook

The bad side is the price of each update. Updating a page means multiplying two whole tables of slopes together, and the size of that work grows with the square of the batch size and with the square of the number of features in each example. For a batch of photos, where every example has many thousands of pixel values, the cost explodes, and it explodes again for every earlier page that has to be updated. Like the knob-nudging experiment, forward mode is correct, and too slow for the models on your phone. Left out here are the exact formulas of the updates and the precise cost estimate; they are in the section on forward-mode differentiation.

Remember

A model's run is a chain of steps that always ends in one number, the total error on the batch, and automatic differentiation must find the slope of that number for every set of weights at once. Nudging each knob and re-running works but costs two runs per number, so it serves only as a check, and plain symbolic differentiation wastes effort on repeated pieces. Forward mode walks the chain in running order, updating a page of slopes for every earlier set of weights at each step. It needs little memory, but every update multiplies whole tables, a cost that explodes for large batches and images, so a better order is needed.

Questions

1. Why does the chain of steps in training always end in one single number?
 - a. Because a computer can only store one number at the end of a program
 - b. Because one number, the total error, is what an optimiser can make smaller
 - c. Because every primitive in the chain outputs a single number
 - d. Because the last step of the model is always a cross-check
2. Why is nudging each knob and re-running the model only good as a spot check?
 - a. Because it gives wrong slopes for most knobs

- b. Because it only works for models without weights
 - c. Because it needs two runs for every single number, millions of runs per training step
 - d. Because it cannot be used on a batch of photos
3. What is wasteful about plain symbolic differentiation?
- a. The derivative formula repeats pieces, like the same sine and cosine, that should be computed once
 - b. It gives only an approximate derivative
 - c. It needs a new run of the model for every weight
 - d. It cannot handle a formula with a sine in it
4. Forward mode needs little memory. So why is it too slow for a photo model?
- a. Because it has to run the whole model once per knob
 - b. Because each update multiplies whole tables, with a cost that grows with the square of batch size and of features
 - c. Because it walks the chain backwards and must store every step
 - d. Because it can only handle one example at a time

Answers

- 1. b)** The last link measures the error on each example and a final step adds them up; one number is what an optimiser can make smaller.
- 2. c)** Numerical differentiation costs two runs per number, so with millions of knobs one training step would need millions of complete runs.
- 3. a)** The derivative formula repeats pieces of the original, and finding the best way to share such work is a problem with no known fast general method.

4. b) Updating a page means multiplying two whole tables of slopes, and that work grows with the square of the batch size and of the number of features, and repeats for every earlier page.

Worked examples

Counting runs for knob-nudging Numerical differentiation needs two runs of the model for every single number whose slope we want. How many complete runs would one training step need for a model with 1,000,000 weights?

Each weight is one number whose slope we need, so there are 1,000,000 slopes to find.

Each slope costs two runs of the whole model.

So the total is $2 \times 1,000,000$ runs.

Answer: 2,000,000 complete runs for a single training step

Practice

1. In forward mode, the cost of each update grows with the square of the batch size. If you double the batch of photos from 32 to 64, by what factor does the cost of one update grow?

Hint: Squaring: what happens to the square of a number when the number doubles?

Answer: By a factor of 4: the cost goes with the square of the batch size, and doubling a number makes its square four times larger ($64^2 / 32^2 = 4$).

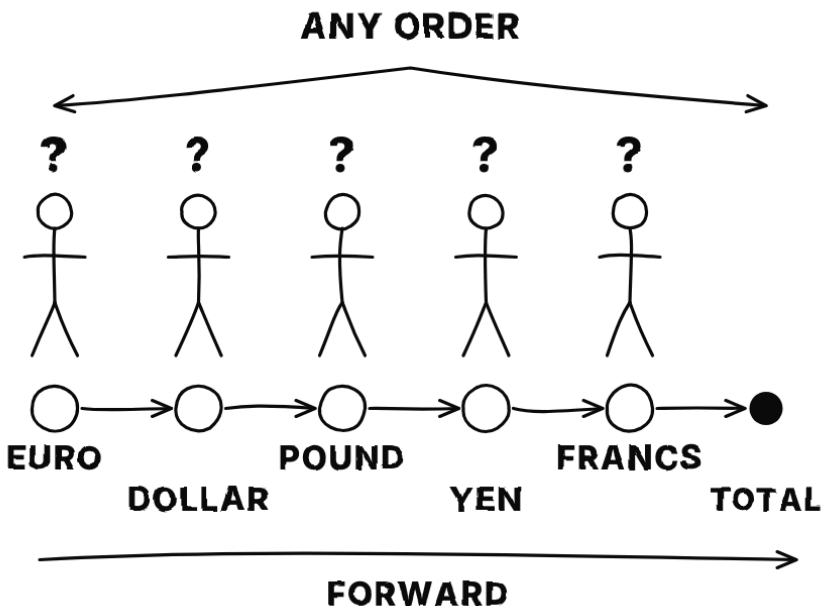
2. In your own words, explain the notebook picture of forward mode: what is written on a new page, and why must earlier pages be updated at every step?

Hint: Think of what each new step does to the influence of all earlier weights on the result.

Answer: When a new piece runs, a fresh page records how its output depends on its own weights. Every earlier page must be updated too, because the new piece passes on the influence of all earlier weights, so their effect on the result changes at each step.

Walking the Chain Backwards

Line up five friends and play a game of exchange rates. The first turns your euros into dollars, the next turns dollars into pounds, then yen, then francs, and the last tells you how many francs one euro finally becomes. You can multiply the five rates in any order and get the same answer, because multiplication does not care about order. But suppose you want to know, for every friend along the line, how much a change in their rate would move the final number; then the order of the work suddenly matters a great deal.

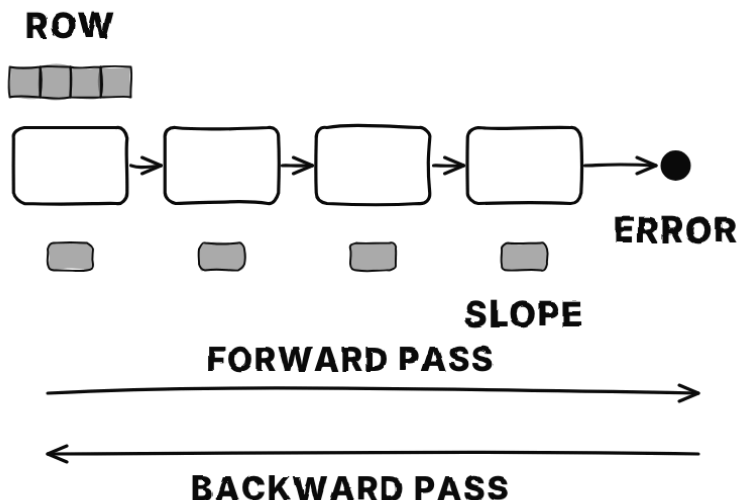


Five Friends, Five Rates

This is the turn the book takes in its section on reverse mode. Written out, the slope of the final error with respect to one set of weights is a long row of tables of slopes multiplied together: the tables of all the later steps, and at the far end the weight table of that one step.

Because multiplication can be grouped in any order, we are free to choose. Forward mode, as we saw, worked in the order the program ran, and paid for it by multiplying whole tables again and again.

The book's key observation is that the chain ends in one number, the total error. So if you start the multiplication from that end, what you carry is never a whole table, only a single row of numbers, and a row times a table is cheap and gives another row. Better still, the row for one step is simply the row of the next step, multiplied once more; so you can build them all by walking the chain backwards, one step at a time. That is reverse-mode automatic differentiation, back-propagation. First the whole program runs forward and keeps every intermediate result; then a mirror-image program walks back from the error to the first step, and at every step it hands out the slope for that step's weights and passes the row further back. The rows it carries, the book calls adjoints: each says how strongly the final error responds to one intermediate result.

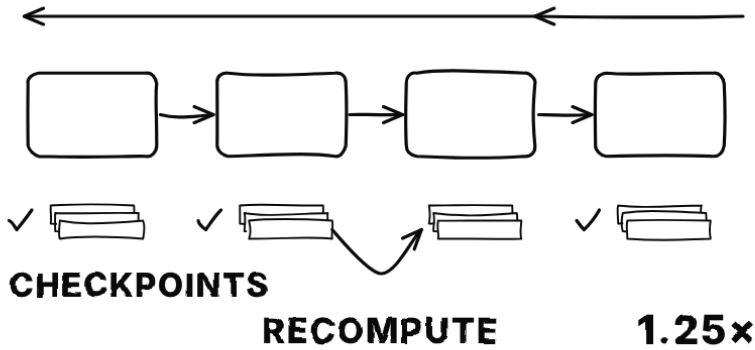


Carrying One Row Back

In neural-network language the ordinary run is the forward pass, not to be confused with forward mode, and the walk back is the backward pass. Every operation in the backward pass is a row times a table, whose cost grows only in proportion to the sizes involved, so reverse mode is far faster than forward mode. When the assistant on your phone was trained, every one of its countless training steps was one forward pass and one such backward pass.

The price is memory. The backward pass needs every intermediate result of the forward pass, so all of them must be kept until the walk back reaches them. Gradient checkpointing eases this by keeping only a few results, the checkpoints, and recomputing the rest when they are needed. The book's figure shows a chain of four blocks: only the

first, second and fourth outputs are kept; the backward pass stops at the third, runs forward again from the nearest checkpoint to rebuild what was missing, and then goes on, at a quarter more computation in this example. The fewer checkpoints you keep, the more you recompute: memory is traded for time. Left out here are the formulas of the reverse program and the step-by-step algorithm; they are in the section on reverse-mode differentiation.



Checkpoints: Memory for Time

Remember

The slope of the error for any weights is a long product of tables, and the order of that product is ours to choose. Since the chain ends in one number, starting from the end means carrying only a row, which is cheap, and building each row from the next one walking backwards. That is back-propagation: a forward pass that keeps every intermediate result, then a backward pass that hands out every slope. It is much faster than forward mode but needs a lot of memory, which checkpointing reduces by recomputing some results instead of keeping them.

Questions

1. Why does starting the multiplication from the end of the chain make reverse mode cheap?
 - a. Because the last tables of the chain are always the smallest ones
 - b. Because the chain ends in one number, so what is carried is only a row, and a row times a table is cheap
 - c. Because the backward pass skips the steps that have no weights
 - d. Because multiplying from the end gives a more accurate answer
2. What must happen before the backward pass can start?

- a. The whole program must run forward and keep every intermediate result
 - b. Every weight must be nudged once to check its slope
 - c. The tables of slopes of all steps must be multiplied together
 - d. The checkpoints must be chosen by the user for every step
3. What does gradient checkpointing trade?
- a. Accuracy of the slopes for speed of the forward pass
 - b. Speed of the backward pass for a smaller final error
 - c. Memory for extra computation: fewer results kept, more recomputed
 - d. The forward pass for a second backward pass
4. In the exchange-rate game, why does the order of work matter if multiplication does not care about order?
- a. Because exchange rates change while you multiply
 - b. Because only the last friend's rate affects the final number
 - c. Because the final number is different depending on the order
 - d. Because you want the effect of every friend's rate, and some orders let you reuse work far better than others

Answers

- 1. b)** The chain ends in one number, the total error, so starting from that end you carry only a single row, and a row times a table is cheap and gives another row.
- 2. a)** First the whole program runs forward and keeps every intermediate result; only then does the mirror-image program walk back from the error.

3. c) Checkpointing keeps only a few results and recomputes the rest when needed; the fewer checkpoints, the more is recomputed, so memory is traded for time.

4. d) The answer is the same in any order, but when you need the influence of every step, choosing the order decides how much work is repeated.

Worked examples

The cost of one checkpoint gap A chain has four blocks. With checkpointing, only the outputs of the first, second and fourth blocks are kept. During the backward pass, the output of the third block is missing and must be rebuilt by running forward again from the nearest checkpoint. How much more computation is needed compared with keeping everything?

The ordinary forward pass runs all four blocks once.

The backward pass stops at the third block, whose result was not kept.

Running forward again from the nearest checkpoint, the second block's output, means running the third block once more.

So there is one extra block's worth of work on top of four, that is five blocks' worth instead of four.

Answer: 1.25 times the computation, a quarter more, in exchange for not keeping one block's result

Practice

1. If you kept no checkpoints at all except the input, what would happen to memory and to computation during the backward pass?

Hint: Ask what must be rebuilt every time the backward pass needs a missing result.

Answer: Memory would be very small, but every missing result would have to be rebuilt by running forward from the start, so the computation

would grow a lot. The fewer checkpoints you keep, the higher the cost of the backward pass.

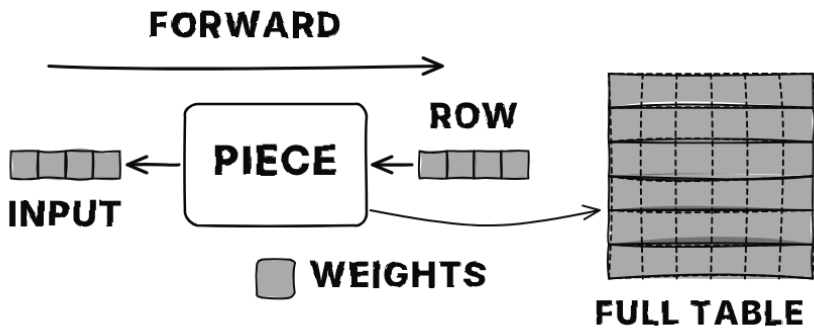
2. Explain in your own words what an adjoint is and why each one can be built from the next one along the chain.

Hint: Think of what the row carried backwards measures, and what one more step back adds to it.

Answer: An adjoint says how strongly the final error responds to one intermediate result. Going one step further back only means multiplying by that step's table of slopes once more, so each adjoint comes from the one after it, and walking backwards builds them all.

Every Piece Learns to Answer Backwards

Ask a shop assistant how the price of a cake would change if flour cost one cent more. They do not need to show you the whole costing spreadsheet of the bakery; a good one just answers, "about half a cent," because they know what goes into the cake. That is the spirit of this section: during the walk back, nobody needs to see the full tables of slopes, only the answer each piece gives when handed the row coming back.



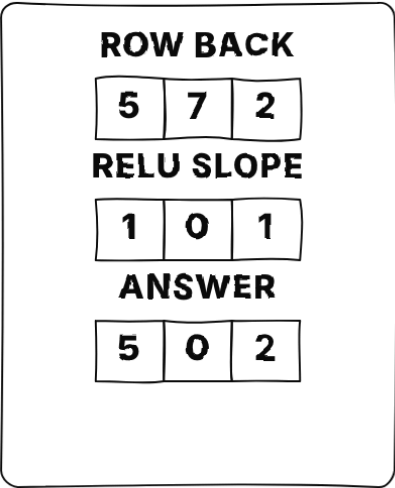
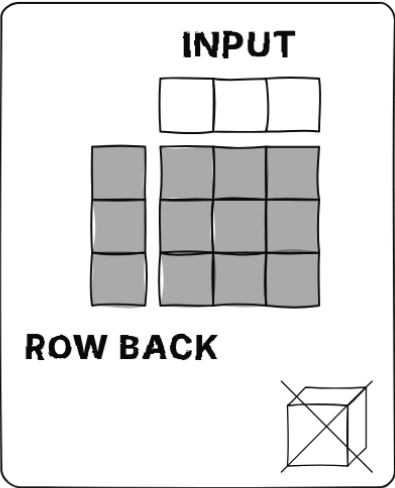
Two Answers Going Back

Look again at what the backward pass actually does at every step. It takes the row it carries, the adjoint, and multiplies it by one table of the current piece: once by the table for the piece's weights, to get their slope, and once by the table for the piece's input, to get the row for the next step back. A row times a table: that is all. The book gives this operation a name, the vector-Jacobian product, and each piece has two of them, one for its input and one for its weights.

From this follows the main practical result. To build a whole system of automatic differentiation, you choose a set of primitives and teach each one, besides its ordinary forward job, how to answer these two backward questions. The full tables of slopes never have to be written out in memory at any point. The book's figure shows a primitive as a box with its forward arrow going one way and two small backward answers coming out the other way, one for the input and one for the weights. And nothing is lost: if you ever do want the full table, you can rebuild it one row at a time by asking the backward question with simple test rows.

Two examples show why the answers are so much cheaper than the tables. For the most common piece of a network, multiplying the input by a table of weights, the full table of slopes with respect to the weights would be a huge three-dimensional block of numbers. Yet the backward answers are simple: the row passed back is just the incoming row run through the same weights turned on their side, and the slope for the weights is a plain grid made from two lists, the incoming row and the input, every entry one product of a pair. For the switch-like pieces between layers, the activation functions such as the ReLU, which act on each number separately, the full table would be almost entirely zeros; the backward answer is simply to scale each number of the incoming row by that piece's own slope at that point.

Every time the photo model in your phone was trained, its backward pass was a long sequence of exactly these small answers. Left out here are the formulas themselves and the short derivation of the weight answer; they are in the subsection on vector-Jacobian products.



Cheap Answers, Two Examples

Remember

Every step of the backward pass is the same move: the row carried back times one table of the current piece. These moves are called vector-Jacobian products, and each piece needs two, one for its input and one for its weights. So an automatic differentiation system is a set of pieces that each know how to answer these two backward questions, and the full tables of slopes are never stored. For a weight multiplication the answers are the weights turned on their side and a grid of products; for an activation, a simple scaling of each number.

Questions

1. Why can an automatic differentiation system work without ever writing out the full tables of slopes?
 - a. Because the tables are all zeros except on the first row
 - b. Because each piece only needs to answer what the incoming row times its table is, and that answer can be computed directly
 - c. Because the backward pass only needs the slope of the last piece
 - d. Because the tables are computed once and then thrown away before training
2. How many backward answers does a piece with weights need to give, and what are they for?

- a. One, the slope of the final error
 - b. Three, one for the input, one for the weights and one for the output
 - c. Two: one for its input, giving the row for the next step back, and one for its weights, giving their slope
 - d. None, the framework computes them from the forward pass alone
3. What is the backward answer of an activation function like the ReLU?
- a. Scale each number of the incoming row by the piece's own slope at that point
 - b. Multiply the incoming row by a huge table of weights
 - c. Turn the incoming row on its side and pass it on unchanged
 - d. Make a grid of products from the incoming row and the input

Answers

- 1. b)** Every step of the backward pass is the row carried back times one table, and each piece can answer that directly, so the full tables are never stored.
- 2. c)** The backward pass multiplies the row once by the weight table, for the weights' slope, and once by the input table, for the row passed further back.
- 3. a)** An activation acts on each number separately, so its full table is almost all zeros and the answer is a simple scaling of each number.

Worked examples

The grid of products for the weights A piece multiplies its input by a table of weights. The input is the list (1, 4) and the row coming

back is (2, 3). The book shows that the slope for the weight in row i and column j is the i -th number of the incoming row times the j -th number of the input. Build the grid of slopes for the weights.

Each entry is one product of a pair: the incoming row number for its row, the input number for its column.

First row, with incoming 2: $2 \times 1 = 2$ and $2 \times 4 = 8$.

Second row, with incoming 3: $3 \times 1 = 3$ and $3 \times 4 = 12$.

No three-dimensional block was needed, only these four products.

Answer: The grid is 2, 8 in the first row and 3, 12 in the second

Practice

1. A ReLU piece received the inputs $(-1, 2, 5)$. Its slope is 0 where the input was negative and 1 where it was positive. The row coming back is (4, 7, 2). What row does the ReLU pass further back?

Hint: Scale each number of the incoming row by the ReLU's slope at the matching input.

Answer: (0, 7, 2): the first input was negative, so its slope is 0 and the 4 becomes 0; the other two slopes are 1, so 7 and 2 pass unchanged.

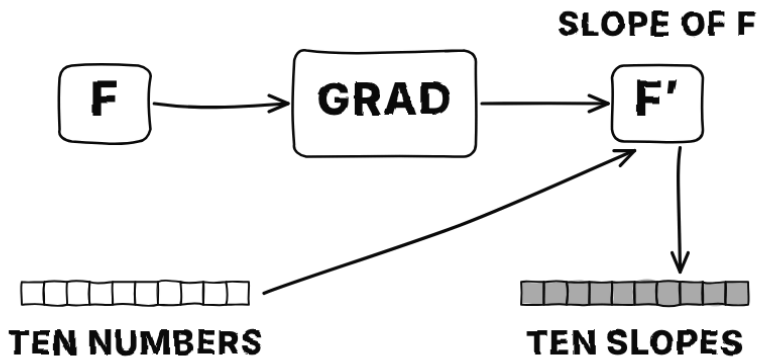
2. If someone still wanted the full table of slopes of a piece, how could they get it using only its backward answer?

Hint: Think of asking the backward question with very simple test rows.

Answer: Ask the backward question once for each simple test row that has a single 1 and zeros elsewhere; each answer is one row of the full table, so the table is rebuilt one row at a time.

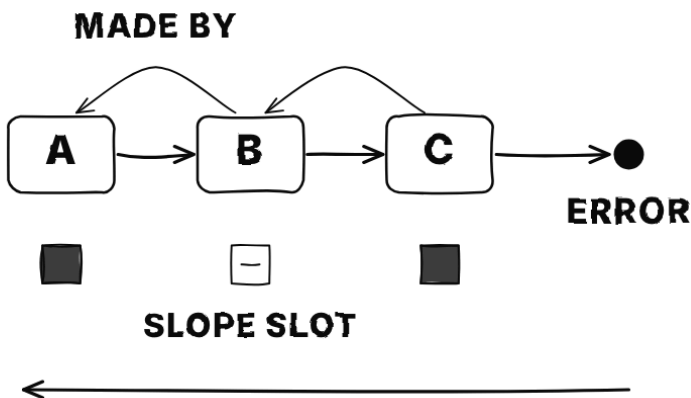
How Libraries Do It, and Why the ReLU Won

Open the calculator on your phone and type one quarter, 0.25. Multiply it by 0.25, and again, and again, ten times in all. You might expect a small number; what you get is about one in a million, and after twenty such steps it is about one in a million million. Keep that collapse in mind, because it decides which activation functions a deep network can use, and it closes this part.



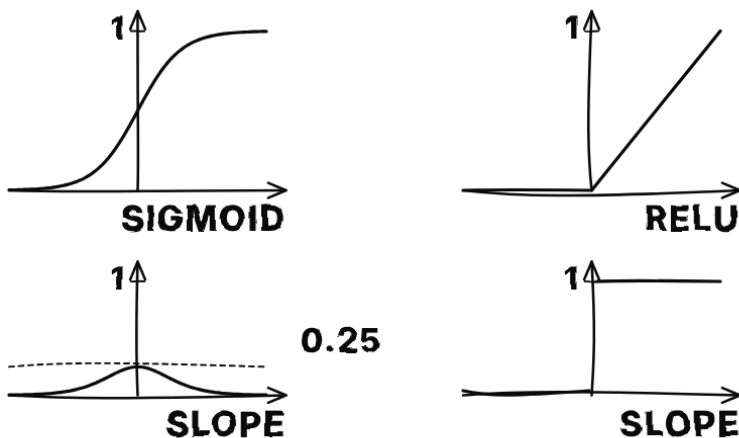
First, the book looks at how real libraries organise the backward pass. There are several ways to build it, from keeping a running list of every operation, as TensorFlow does, to rewriting the program's own code into a backward version. In a function-style library such as JAX, the idea of the previous part becomes very literal: the backward answer is a function that takes a function and a point, and hands back a new function waiting for the incoming row. Likewise the gradient itself is a function that takes your function and returns another function, which computes the slope at any point you give it. The book's small example does exactly this for the sum of squares of ten numbers: you ask for its gradient, and you get back a function whose answer, at any ten numbers, is again a list of ten slopes. PyTorch offers the same style in a module of its own, and by combining these function-makers one can compute more advanced things, such as slopes of slopes.

Object-style models, the usual way of writing PyTorch, need one more step. One option is to pull the weights out of the model object and call it as a plain function with those weights handed in. More commonly PyTorch does the work behind the scenes: every result it computes remembers which operation created it, and it has a slot for its own slope, empty at first. When you ask for the backward pass on the final error, PyTorch follows those memories back through the chain and fills the slots of every quantity that asked to keep a slope; only those, so that memory is not wasted. Every time you see a model being fine-tuned on a laptop, that call to the backward pass is doing exactly this walk.



Results That Remember

Now back to the calculator. In the backward pass, every activation function multiplies the row going back by its own slope. If that slope is below one everywhere, a deep network multiplies the row by a small number again and again, and the slopes of the early layers shrink towards zero: the vanishing gradient problem. If it is above one everywhere, they grow without bound: the exploding gradient problem. These are not only theoretical, because computers store numbers with limited precision, so numbers that become too small or too large are simply lost after not many layers. A curious aside in the book: precisely because of that limited precision, a stack of purely linear layers is not perfectly linear, and this has even been used to train such networks.



Sigmoid Shrinks, ReLU Passes

The sigmoid, a smooth S-shaped curve that was popular in the past, shows the trouble at its worst: its slope is never more than one quarter, the number you just multiplied by itself. The book's figure draws the S-curve together with its slope, a low hump that peaks at a quarter in the middle and flattens to nothing on both sides. A perfect activation would have slope exactly one everywhere, but that is the function that does nothing at all, and does not separate one layer from the next. The ReLU is the compromise: for negative inputs its slope is zero, so the row is simply switched off there, and for positive inputs its slope is one, so the row passes unchanged, with no shrinking and no growing. Its figure is a flat line that turns into a straight ramp, and its slope a step from zero up to one. One more gift: since the ReLU only blanks out negatives, its slope can be read from its output as well as

from its input, so the input need not be kept, and libraries can overwrite it in place to save memory. Left out here are the exact backward formulas and code of these libraries; the subsection on implementing reverse mode gives them.

Remember

Libraries build the backward pass in different ways: JAX turns gradients into functions that return functions, and PyTorch has every result remember the operation that made it and fill its own slope slot during the backward pass. In the backward pass each activation multiplies the row by its slope, so a slope below one everywhere makes gradients vanish in deep networks and one above one makes them explode. The sigmoid's slope is at most one quarter, which is why it fails in deep networks. The ReLU passes the row either unchanged or not at all, which keeps gradients healthy, and it even lets libraries save memory by overwriting its input.

Questions

1. Why do deep networks built with the sigmoid suffer from vanishing gradients?
 - a. Because the sigmoid's slope is at most one quarter, so the row going back shrinks at every layer
 - b. Because the sigmoid has no slope at all for negative inputs

- c. Because the sigmoid's slope is above one, so the numbers overflow
 - d. Because the sigmoid cannot be computed with limited precision
2. Why not use an activation whose slope is exactly one everywhere?
- a. Because computers cannot store the number one exactly
 - b. Because such a function would make gradients explode
 - c. Because the only such function does nothing and does not separate one layer from the next
 - d. Because its backward answer would need a full table of slopes
3. How does PyTorch know where to go during the backward pass of an object-style model?
- a. It runs the model again in reverse order of the code lines
 - b. Every result remembers the operation that created it, and the backward pass follows those memories back
 - c. It asks the user to list the operations of the model
 - d. It turns the model into a list of weights and differentiates each one by nudging
4. Why can a library overwrite the input of a ReLU without breaking the backward pass?
- a. Because the ReLU has no backward answer
 - b. Because the ReLU's slope is always one
 - c. Because the backward pass never visits activation functions

- d. Because the ReLU only blanks out negatives, so its slope can be read from its output as well

Answers

1. **a)** Each activation multiplies the row going back by its slope, and the sigmoid's slope is never more than one quarter, so many layers shrink the slopes of early layers towards zero.
2. **c)** A slope of one everywhere means the function that does nothing at all, which does not separate the layers; the ReLU is the compromise.
3. **b)** Every result PyTorch computes remembers which operation created it, and the backward pass follows those memories back through the chain, filling the slope slots.
4. **d)** The ReLU keeps positive values unchanged and blanks out negatives, so its slope is the same whether read from the input or the output, and the input need not be kept.

Worked examples

Ten sigmoid layers at best The sigmoid's slope is never more than one quarter. In the backward pass, every sigmoid layer multiplies the row going back by its slope. Even in the best case, by what factor is a slope shrunk after passing back through ten sigmoid layers?

The best case is the largest slope, one quarter, at every layer.

Each layer multiplies by 0.25, so ten layers multiply by 0.25 ten times over.

0.25 to the power ten is about 0.00000095, roughly one in a million.

Answer: At least about a millionfold, even in the best case

Practice

1. A row coming back passes through five ReLU layers, and at every layer the input of the number we follow was positive. By what factor has that number been multiplied?

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

Answer: By exactly one: the ReLU's slope for positive inputs is one, and one multiplied by itself five times is still one, so the number passes back unchanged, with no shrinking and no growing.

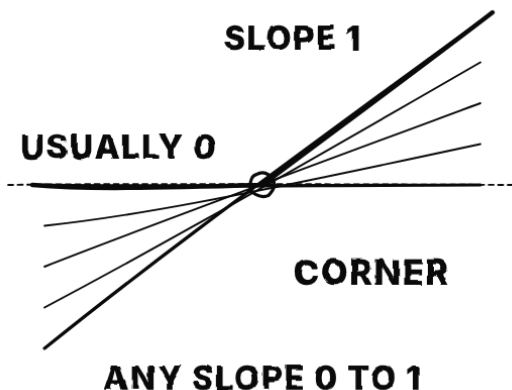
2. The book's example asks for the gradient of the sum of the squares of ten numbers. When you call the gradient function at some ten numbers, how many numbers does it give back, and why?

Hint: The gradient holds one slope for each number the function takes in.

Answer: Ten: the gradient gives one slope for each of the ten input numbers, so the answer is again a list of ten.

The Corner of the Ramp

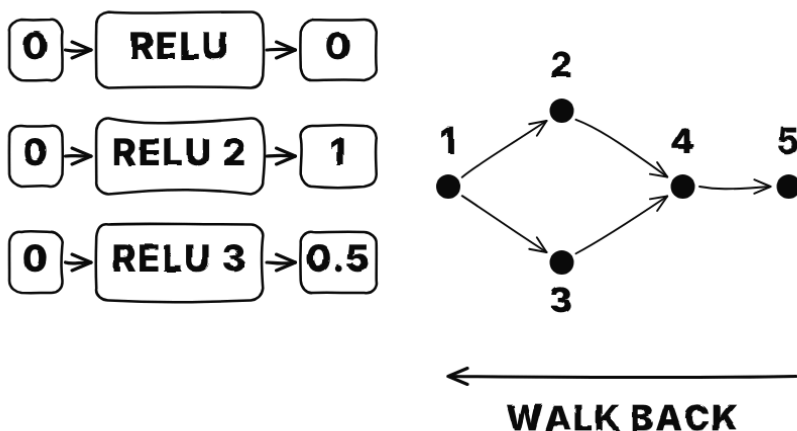
Lay a ruler on the floor against the foot of a ramp, where the flat floor meets the slope. You can tilt the ruler so it lies flat along the floor, or tilt it up along the ramp, or anywhere in between, and in every position it still touches the corner while staying entirely under the floor and the ramp. On the smooth floor, or on the smooth ramp, the ruler has only one way to lie; at the corner it has a whole fan of them. That is exactly the trouble with the ReLU, whose graph is flat floor and then a straight ramp, and it is the last question of the chapter: what slope do we use at the corner?



A Fan of Rulers at the Corner

The practical answer comes first. Training starts from random numbers and moves in small, noisy steps, so the chance of landing exactly on the corner, at an input of exactly zero, is practically nil; everywhere else the slope is perfectly defined. The technical answer is the subgradient: any slope of a line that touches the curve at a point and stays below the whole curve, like the ruler. Where a curve is smooth there is just one such line, the ordinary derivative; at a corner there are many, and together they form the subdifferential. For the ReLU at zero, any slope from zero up to one will do, and most libraries simply pick zero. Training that picks such a slope at each step is called subgradient descent.

The book adds a warning. For curves that bend both ways, subgradients need not exist, and mathematicians use a looser, local version instead. Worse, two programs that compute exactly the same function can report different slopes at a corner: the book cites a case where three ways of writing the ReLU give zero, one and one half in PyTorch at the same point.



Same Function, Different Slopes

The chapter closes with practice. The book recommends building a tiny reverse-mode system yourself, and points to micrograd, a small teaching version; once chains branch and merge, the steps must first be put in a proper order before walking back. It also suggests writing a new primitive with its own backward answers, in PyTorch or in JAX, and reading the JAX guide to automatic differentiation for slopes of slopes and more. Left out here are the formal definitions and the details of these exercises; they are in the subsection on subdifferentiability and in the closing section. With that, the whole machinery behind the training of every chat assistant and photo app is in place: one error at the end, and a fast walk back that tells every knob which way to turn.

Remember

The ReLU has a corner at zero, where the ordinary slope does not exist. In practice training almost never lands exactly there, and in theory any slope of a line touching the corner from below, a subgradient, may be used; for the ReLU any value from zero to one, usually zero. Corners can still make different but equivalent programs report different slopes. The best way to understand all of this is to build a small reverse-mode system yourself.

Questions

1. Why is the corner of the ReLU rarely a problem when training in practice?
 - a. Because libraries remove the corner before training starts
 - b. Because training starts from random numbers and moves in noisy steps, so landing exactly on zero is practically impossible
 - c. Because the ReLU's slope at zero is always one
 - d. Because the backward pass skips the ReLU layers
2. At a smooth point of a curve, how many subgradients are there?
 - a. Exactly one, the ordinary derivative
 - b. None, subgradients exist only at corners
 - c. Two, one from each side
 - d. A whole range from zero to one

3. Which slopes are valid subgradients of the ReLU at zero?
 - a. Only zero
 - b. Only one
 - c. Any value from zero up to one, with libraries usually picking zero
 - d. Any value at all, since the slope is undefined there
4. What does the book's example of three ways of writing the ReLU show?
 - a. That two of the three ways compute a different function
 - b. That PyTorch cannot differentiate the ReLU at all
 - c. That subgradient descent never works for the ReLU
 - d. That programs computing the same function can report different slopes at a corner

Answers

1. **b)** Training starts from random numbers and moves in small, noisy steps, so the chance of an input of exactly zero is practically nil, and everywhere else the slope is defined.
2. **a)** Where a curve is smooth there is just one line touching it from below, the ordinary derivative; only at a corner are there many.
3. **c)** For the ReLU at zero any slope from zero up to one gives a line that touches the corner and stays below the curve, and most libraries pick zero.
4. **d)** Three ways of writing the same ReLU give zero, one and one half in PyTorch at the same point: corners can make equivalent programs report different slopes.

Worked examples

Same function, different slopes A second ReLU is written as: the ReLU of minus s , plus s . A third is half of the sum of the ordinary ReLU and the second one. PyTorch uses slope 0 for the ordinary ReLU at zero. What slope does the backward pass report at zero for the second and the third?

The second function equals the ReLU everywhere: for positive s it gives $0 + s = s$, for negative s it gives minus s plus $s = 0$.

Its backward pass at zero adds the slope of the ReLU part, taken as 0, and the slope of plus s , which is 1: total 1.

The third is half of (ReLU + second ReLU), so its reported slope is half of $(0 + 1)$.

So the three equal functions report 0, 1 and 0.5 at the same point.

Answer: 1 for the second, 0.5 for the third, against 0 for the ordinary ReLU

Practice

1. Is a slope of 0.3 a valid subgradient of the ReLU at zero? And a slope of 1.5?

Hint: Picture a line through the corner with that slope: does it stay below both the floor and the ramp?

Answer: 0.3 is valid, since it lies between 0 and 1 and its line stays under both the floor and the ramp. 1.5 is not: for positive inputs its line rises faster than the ramp and goes above the curve.

2. In your own words, explain why a home-made reverse-mode system must put the steps in a proper order before walking back once the chain branches and merges.

Hint: Think of a quantity used by two later steps: when is its row coming back complete?

Answer: When chains branch and merge, a quantity can feed several later steps, and its backward row is complete only once all of them have sent their contribution back. Ordering the steps first makes the walk back visit each quantity only after everything that depends on it, so no wrong path is taken.