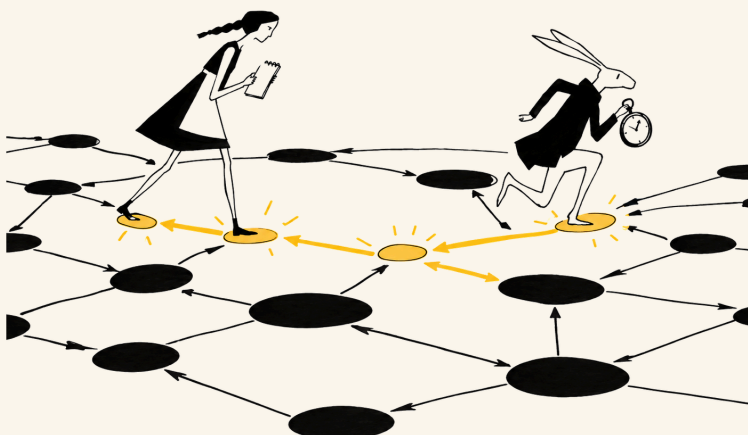


Alice's Adventures in a Differentiable Wonderland



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

Automatic differentiation

Problem statement

Forward-mode differentiation

Reverse-mode differentiation

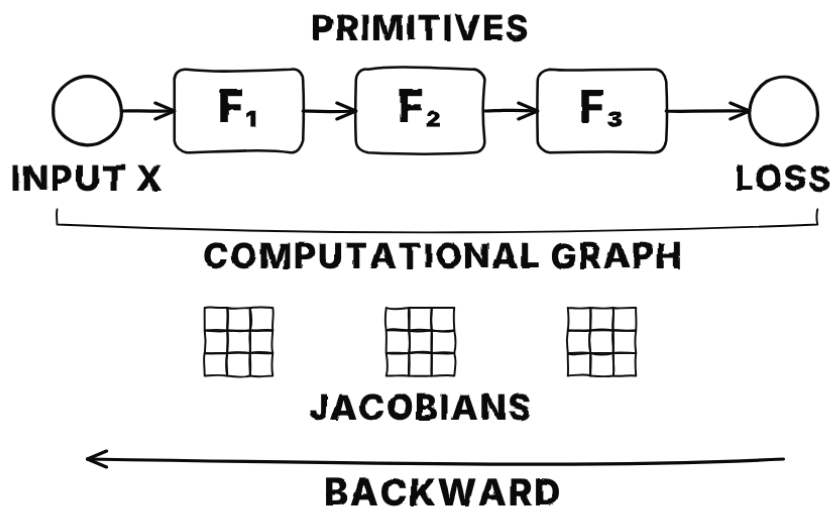
Vector-Jacobian products

Implementing a reverse-mode system

Choosing an activation function

Automatic differentiation

The previous chapter highlighted the need for an efficient, automatic procedure to compute gradients of any possible sequence of operations. In this chapter we describe such a method, called back-propagation in the neural network literature, or reverse-mode automatic differentiation in the computer science one. Its analysis has several insights, ranging from the choice of the model to the memory requirements for optimizing it.



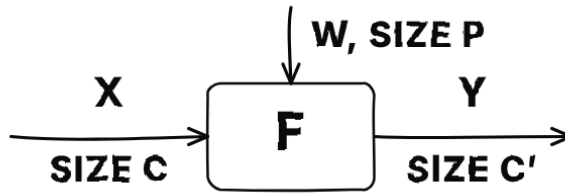
A Program as a Chain of Primitives

Problem setup.

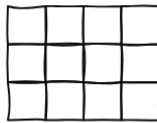
We consider the problem of efficiently computing gradients of generic computational graphs, such as those induced by optimizing a scalar loss function on a fully-connected model, a task called automatic differentiation, or AD. You can think of a computational graph as the set of atomic operations, which we call primitives, obtained by running the program itself. We will consider sequential graphs for brevity, but everything can be easily extended to acyclic and even more generic computational graphs.

The problem may seem trivial, since the chain rule of Jacobians, from the section on gradients and Jacobians, tells us that the gradient of a function composition is simply the matrix product of the corresponding Jacobian matrices. However, efficiently implementing this process is the key challenge of this chapter, and the resulting algorithm, reverse-mode AD or backpropagation, is a cornerstone of neural networks and of differentiable programming in general. Understanding it is also key to understanding the design, and the differences, of most frameworks for implementing and training such programs, such as TensorFlow, PyTorch or JAX.

To set up the problem, we assume we have at our disposal a set of primitives, $y = f_i(\mathbf{x}, \mathbf{w}_i)$. Each primitive represents an operation on an input vector $\mathbf{x} \sim (c_i)$, parameterized by the vector $\mathbf{w}_i \sim (p_i)$, for example the weights of a linear projection, and giving as output another vector $\mathbf{y} \sim (c'_i)$.



INPUT JAC.



C' x C

WEIGHT JAC.



C' x P

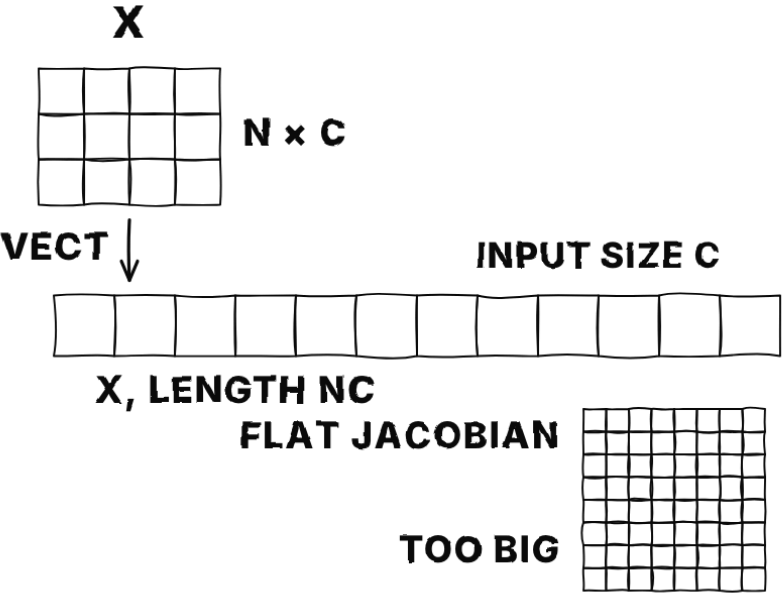
One Primitive and Its Two Jacobians

There is a lot of flexibility in our definition of primitives, which can represent basic linear algebra operations such as matrix multiplication, layers in the sense of Chapter 5, such as a fully-connected layer with an activation function, or even larger blocks or models. This recursive composability is a key property of programming and extends to our case.

We only assume that for each primitive we know how to compute the partial derivatives with respect to its two input arguments, which we call the input Jacobian and the weight Jacobian of the operation. The input Jacobian is $\partial_{\mathbf{x}}[f(\mathbf{x}, \mathbf{w})] \sim (c', c)$. The weight Jacobian is $\partial_{\mathbf{w}}[f(\mathbf{x}, \mathbf{w})] \sim (c', p)$.

These are reasonable assumptions, since we restrict our analysis to differentiable models. Continuous primitives with one or more points of non-differentiability, such as the ReLU, can be made to fit into this framework with the use of subgradients, as we will see at the end of this chapter. Non-differentiable operations such as sampling or thresholding can also be included, by finding a relaxation of their gradient or an equivalent estimator. We cover the latter case in the next volume.

On our notation and higher-order Jacobians.



Flattening a Batch into a Vector

We only consider vector-valued quantities for readability, as all resulting gradients are then matrices. In practice, existing primitives

may have inputs, weights or outputs of higher rank. For example, consider a basic fully-connected layer on a mini-batched input, $f(\mathbf{X}, \mathbf{W}) = \mathbf{X}\mathbf{W} + \mathbf{b}$. In this case the input $\mathbf{X}$ has shape (n, c) , the weights have shapes (c, c') and (c') , with c' a hyper-parameter, and the output has shape (n, c') . Hence the input Jacobian has shape (n, c', n, c) , and the weight Jacobian has shape (n, c', c, c') , both having rank four.

In our notation, we can consider the equivalent flattened vectors $\mathbf{x} = \text{vect}(\mathbf{X})$ and $\mathbf{w} = [\text{vect}(\mathbf{W}); \mathbf{b}]$, and our resulting flattened Jacobians have shapes (nc', nc) and (nc', cc') respectively. This is crucial in the following, since every time we refer to “the input size c ” we are referring to “the product of all input shapes”, including any mini-batching dimensions. This also shows that, while we may know how to compute the Jacobians, we may not wish to fully materialize them in memory, due to their large dimensionality.

As a final note, our notation aligns with the way these primitives are implemented in a functional library, such as JAX. In an object-oriented framework, such as TensorFlow or PyTorch, we saw in the previous chapter that layers are implemented as objects, with the parameters being a property of the object, and the function call being replaced by a method of the object. This style simplifies certain practices, such as deferring the initialization of all parameters until the input shapes are known, called lazy initialization, but it adds a small layer of abstraction to consider when translating our notation into workable code. As we will see, these differences are reflected in turn in the way AD is implemented in the two kinds of frameworks.

Questions

1. Why does the chapter treat computing gradients as a real problem, when the chain rule already says the gradient of a composition is a product of Jacobians?
 - a. Because the chain rule does not hold for matrices
 - b. Because implementing that product efficiently is the hard part
 - c. Because the Jacobians of the primitives are unknown
 - d. Because the chain rule only applies to scalar functions
2. What does the setup assume we know for every primitive?
 - a. Only its output for every possible input
 - b. Its partial derivatives with respect to its input and with respect to its weights
 - c. The gradient of the final loss with respect to its weights
 - d. A symbolic formula for the whole program
3. A fully-connected layer works on a mini-batch with input shape (n, c) and output shape (n, c') . Why might we not want to store its input Jacobian in memory?
 - a. Because it is always zero
 - b. Because flattened it has shape (nc', nc) , which can be very large
 - c. Because it has rank 1 and carries no information
 - d. Because object-oriented frameworks cannot store matrices
4. How can a primitive like the ReLU, which is not differentiable at one point, still fit into this framework?
 - a. By leaving it out of the computational graph
 - b. By replacing it with a linear layer

- c. Through the use of subgradients
- d. By sampling its output at random

Answers

1. **b)** The chain rule gives the answer as a matrix product of Jacobians, but efficiently implementing this process is the key challenge of the chapter.
2. **b)** We only assume that for each primitive we know how to compute the input Jacobian and the weight Jacobian, its partial derivatives with respect to its two arguments.
3. **b)** Its input Jacobian has rank four, and flattened it has shape (nc', nc) ; because of this large dimensionality we may not wish to fully materialize it in memory.
4. **c)** Continuous primitives with points of non-differentiability, such as the ReLU, can be made to fit into this framework with the use of subgradients.

Worked examples

Size of a flattened Jacobian A fully-connected layer receives a mini-batch of $n = 32$ inputs, each with $c = 10$ features, and produces $c' = 5$ outputs per input. How many numbers are in its flattened input Jacobian, and how many in its flattened weight Jacobian (ignoring the bias)?

The flattened input Jacobian has shape (nc', nc) , because every output number is differentiated with respect to every input number.

Here $nc' = 32 \times 5 = 160$ and $nc = 32 \times 10 = 320$, so the shape is $(160, 320)$.

That gives $160 \times 320 = 51,200$ numbers.

The flattened weight Jacobian has shape (nc', cc') , with $cc' = 10 \times 5 = 50$ weights in W .

That gives $160 \times 50 = 8,000$ numbers, far more than the 50 weights themselves.

Answer: 51,200 numbers for the input Jacobian and 8,000 for the weight Jacobian

Practice

1. A layer has input shape (n, c) with $n = 2$ and $c = 3$, and output shape (n, c') with $c' = 4$. Write the rank-4 shape of its input Jacobian, and its flattened shape. Do both contain the same number of entries?

Hint: The rank-4 shape is (n, c', n, c) ; the flattened one multiplies the first two and the last two.

Answer: The rank-4 shape is $(2, 4, 2, 3)$ and the flattened shape is $(8, 6)$. Both contain 48 entries: flattening only rearranges the numbers.

2. In this chapter, “the input size c ” of a primitive means the product of all its input shapes. Why does that matter when a model is trained on mini-batches?

Hint: Think of what happens to the input size when the mini-batch dimension is included.

Answer: The mini-batching dimension is part of the input size, so c grows with the batch. The Jacobians, whose shapes are built from these sizes, grow with it, which is why we may know how to compute them and still not want to store them in memory.

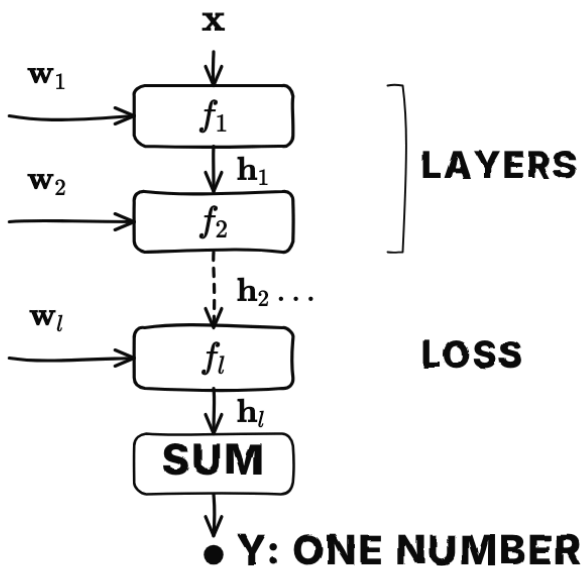
3. Explain in your own words the difference between how a functional library such as JAX and an object-oriented framework such as PyTorch treat a layer's parameters, and how this relates to the notation $f(x, w)$.

Hint: Ask where the weights live in each case.

Answer: In the notation $f(x, w)$, the weights are an explicit second argument of the function, which is how a functional library like JAX treats them. In an object-oriented framework the layer is an object, the parameters are a property of that object, and the function call is a method; this makes things like lazy initialization easier but adds a layer of abstraction between the notation and the code.

Problem statement

With all these details out of the way, we are ready to state the AD task. Consider a sequence of l primitive calls, followed by a final summation. First, $\mathbf{h}_1 = f_1(\mathbf{x}, \mathbf{w}_1)$. Then $\mathbf{h}_2 = f_2(\mathbf{h}_1, \mathbf{w}_2)$, and so on, down to $\mathbf{h}_l = f_l(\mathbf{h}_{l-1}, \mathbf{w}_l)$. Finally, $y = \Sigma \mathbf{h}_i$.



The Evaluation Trace

This is called an evaluation trace of the program. Roughly, the first $l - 1$ operations can represent several layers of a differentiable model, operation l can be a per-input loss, for example cross-entropy, and the final operation sums the losses of the mini-batch. Hence the output of

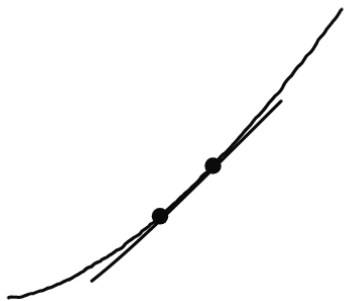
our program is always a scalar, since we require it for numerical optimization. We abbreviate the previous program as $F(\mathbf{x})$.

Definition: automatic differentiation. Given a program $F(\mathbf{x})$ composed of a sequence of differentiable primitives, automatic differentiation, or AD, refers to the task of simultaneously and efficiently computing all weight Jacobians of the program, given knowledge of the computational graph and of all the individual input and weight Jacobians. In symbols, $AD(F(\mathbf{x}))$ is the set of the gradients $\partial_{\mathbf{w}_i} y$, for $i = 1, \dots, l$.

As we will see, there are two major classes of AD algorithms, called forward-mode and backward-mode, corresponding to a different ordering in the composition of the individual operations. We will also see that the backward mode, called back-propagation in the neural network literature, is significantly more efficient in our context. While we focus on a simplified scenario, it is relatively easy to extend our derivation to acyclic graphs of primitives, as already mentioned, and also to situations where parameters are shared across layers, which is called weight sharing. We will see an example of weight sharing in Chapter 13.

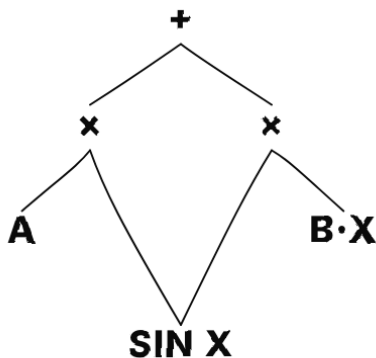
Numerical and symbolic differentiation.

NUMERICAL



2 CALLS

SYMBOLIC



**COMPUTED ONCE
NP-COMPLETE**

Numerical Versus Symbolic Differentiation

Before moving on to forward-mode AD, we comment on the difference between AD and other classes of algorithms for differentiating functions. First, we could directly apply the definition of gradients to obtain a suitable numerical approximation of the gradient. This process is called numerical differentiation. However, each scalar value to be differentiated requires two function calls in a naive implementation, making this approach unfeasible except for numerical checks over the implementation.

Second, consider this simple function: $f(x) = a \sin(x) + b x \sin(x)$. We can ask a symbolic engine to pre-compute the full, symbolic equation of the derivative. This is called symbolic differentiation. In the book's Box 6.1, the Python library SymPy is given this function, and asked for

its derivative with respect to x , it returns a $\cos(x) + b \times \cos(x) + b \sin(x)$.

In a realistic implementation, the intermediate value $h = \sin(x)$ would be computed only once and stored in an intermediate variable, which can also be reused for the corresponding computation in the gradient trace, and a similar reasoning goes for the $\cos(x)$ term in the derivative. This is less trivial than it appears: finding an optimal implementation for the Jacobian which avoids any unnecessary computation is an NP-complete task, known as optimal Jacobian accumulation. However, we will see that we can exploit the structure of our program to devise a suitably efficient implementation of AD that is significantly better than a symbolic approach like the above, and it is, in fact, equivalent to a symbolic approach that allows for the presence of subsequences.

Questions

1. Why does the program in the evaluation trace end with a sum, so that its output y is a single number?
 - a. Because summing makes the Jacobians smaller
 - b. Because numerical optimization requires a scalar output
 - c. Because the last layer of a model is always a sum
 - d. Because forward-mode differentiation needs it
2. What exactly does automatic differentiation have to compute, according to the definition?

- a. All the weight Jacobians of the program, simultaneously and efficiently
 - b. The input Jacobian of each primitive, one at a time
 - c. A symbolic formula for the program's output
 - d. The value of the program for every input
3. Why is numerical differentiation unfeasible for a model, except as a check?
- a. Because it gives the exact derivative only for linear functions
 - b. Because it needs a symbolic engine
 - c. Because each scalar value to be differentiated needs two function calls in a naive implementation
 - d. Because it cannot handle a scalar output
4. What is the weakness of the symbolic derivative of $f(x) = a \sin(x) + b \times \sin(x)$ as written out in full?
- a. It is mathematically wrong
 - b. It repeats work, such as $\sin(x)$ and $\cos(x)$, that a realistic implementation would compute once and store
 - c. It cannot be evaluated by a computer
 - d. It only works when a and b are zero

Answers

1. **b)** The final operation sums the losses of the mini-batch, so the output of the program is always a scalar, which we require for numerical optimization.
2. **a)** Automatic differentiation is the task of simultaneously and efficiently computing all weight Jacobians of the program, given the computational graph and the individual input and weight Jacobians.

3. c) Each scalar value to be differentiated requires two function calls in a naive implementation, which makes the approach unfeasible except for numerical checks.

4. b) In a realistic implementation the intermediate value $h = \sin(x)$ would be computed only once and stored, and the same reasoning goes for the $\cos(x)$ term; avoiding all unnecessary computation is far from trivial.

Worked examples

The symbolic derivative at one point Take the book's function $f(x) = a \sin(x) + b x \sin(x)$, whose symbolic derivative is $a \cos(x) + b x \cos(x) + b \sin(x)$. What is the derivative at $x = 0$ when $a = 2$ and $b = 3$?

Write the derivative the symbolic engine returns: $a \cos(x) + b x \cos(x) + b \sin(x)$.

At $x = 0$, $\cos(0) = 1$ and $\sin(0) = 0$.

The first term is $a \times 1 = 2$.

The second term has the factor $x = 0$, so it is 0; the third has $\sin(0) = 0$, so it is 0 too.

Add the three terms: $2 + 0 + 0 = 2$.

Answer: 2

Practice

1. A model has 1,000,000 weights. Using naive numerical differentiation, which needs two function calls for each scalar value to be differentiated, how many calls of the whole program would one gradient take?

Hint: Each weight is one scalar value to be differentiated.

Answer: 2,000,000 calls: two for each of the million weights, which is why numerical differentiation is used only to check an implementation.

2. Write the evaluation trace for a program with $l = 3$ primitives, ending with the final sum. Which operation could be the per-input loss?

Hint: Each primitive takes the previous output and its own weights.

Answer: $h_1 = f_1(x, w_1)$, $h_2 = f_2(h_1, w_2)$, $h_3 = f_3(h_2, w_3)$, $y = \text{sum of } h_3$.

The first two operations can be layers of the model, the third, operation l , can be the per-input loss such as cross-entropy, and the sum adds up the losses of the mini-batch.

3. Explain in your own words what forward-mode and backward-mode automatic differentiation have in common and where they differ, as far as this part tells.

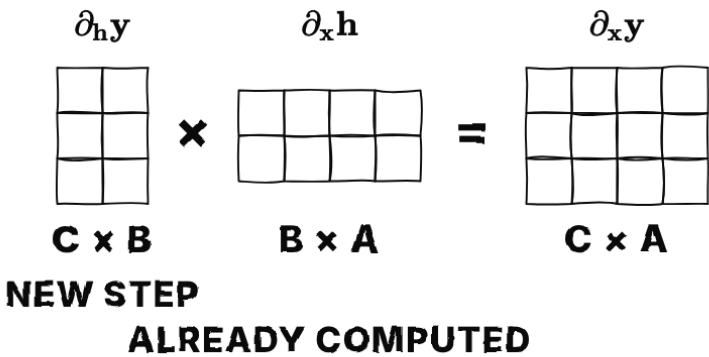
Hint: Both compute the same thing; look at the ordering.

Answer: Both compute all the weight Jacobians of the program from the individual Jacobians of its primitives. They differ in the order in which the individual operations are composed, and the backward mode, called back-propagation, is significantly more efficient in this setting.

Forward-mode differentiation

We begin by recalling the chain rule of Jacobians. Consider a combination of two primitive functions, $\mathbf{h} = f_1(\mathbf{x})$ and $\mathbf{y} = f_2(\mathbf{h})$. In terms of their gradients, we have $\partial_{\mathbf{x}}\mathbf{y} = \partial_{\mathbf{h}}\mathbf{y} \cdot \partial_{\mathbf{x}}\mathbf{h}$.

$$\partial_{\mathbf{x}}\mathbf{y} = \partial_{\mathbf{h}}\mathbf{y} \cdot \partial_{\mathbf{x}}\mathbf{h}$$



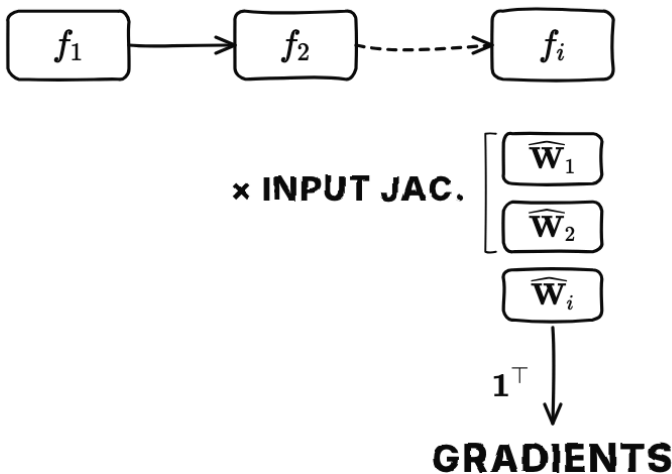
The Chain Rule as Matrix Shapes

If $\mathbf{x}$, $\mathbf{h}$ and $\mathbf{y}$ have dimensions a , b and c respectively, the previous Jacobian requires the multiplication of a $c \times b$ matrix, the Jacobian of $\mathbf{y}$ with respect to $\mathbf{h}$, with a $b \times a$ one, the Jacobian of $\mathbf{h}$ with respect to $\mathbf{x}$. We can interpret the rule as follows: if we have already computed f_1

and its Jacobian, once we apply f_2 we can “update” the gradient by multiplying it with the corresponding Jacobian.

We can immediately apply this insight to obtain a working algorithm called forward-mode automatic differentiation, or F-AD. The idea is that every time we apply a primitive function, we initialize its corresponding weight Jacobian, called the tangent in this context, while simultaneously updating all previous tangent matrices. Let us see a simple worked-out example to illustrate the main algorithm.

Consider the first instruction in our program, $\mathbf{h}_1 = f_1(\mathbf{x}, \mathbf{w}_1)$. Because nothing has been stored up to now, we initialize the tangent matrix for $\mathbf{w}_1$ as its weight Jacobian: $\hat{\mathbf{W}}_1 = \partial_{\mathbf{w}_1} \mathbf{h}_1$.



Forward Mode Carries the Tangents

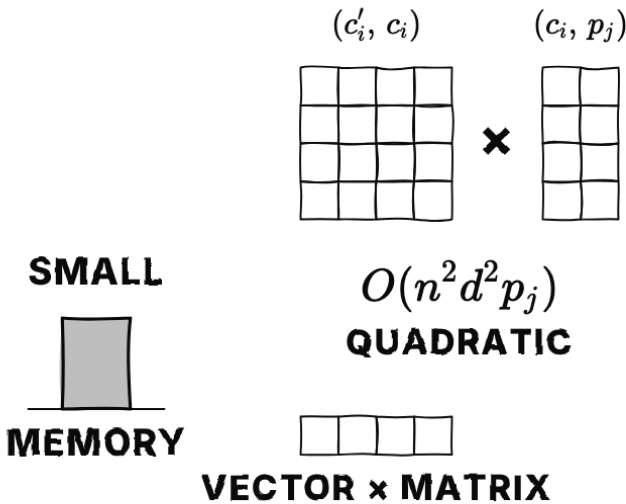
We now proceed to the second instruction, $\mathbf{h}_2 = f_2(\mathbf{h}_1, \mathbf{w}_2)$. We update the previous tangent matrix while simultaneously initializing the second one. The tangent matrix for $\mathbf{w}_1$ is updated by multiplying it by the input Jacobian of f_2 : $\hat{W}_1 \leftarrow [\partial_{\mathbf{h}_1} \mathbf{h}_2] \hat{W}_1$. The second one is initialized as $\hat{W}_2 = \partial_{\mathbf{w}_2} \mathbf{h}_2$.

The update requires the input Jacobian of the primitive, while the second term requires the weight Jacobian of the primitive.

Abstracting away, consider the generic i -th primitive, given by $\mathbf{h}_i = f_i(\mathbf{h}_{i-1}, \mathbf{w}_i)$. We initialize the tangent matrix for $\mathbf{w}_i$ while simultaneously updating all previous matrices. For every j smaller than i , $\hat{W}_j \leftarrow [\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i] \hat{W}_j$. And the new one is the weight Jacobian of f_i , $\hat{W}_i = \partial_{\mathbf{w}_i} \mathbf{h}_i$.

There are $i - 1$ updates in the first row, one for each tangent matrix we have already stored in memory, with the input Jacobian of the i -th operation being shared by all previous tangents. The last operation in the program is a sum, and the corresponding gradient gives us the output of the algorithm: for every i , $\nabla_{\mathbf{w}_i} y = \mathbf{1}^T \hat{W}_i$. To be fully consistent with notation, this output is a row vector, while we defined the gradient as a column vector; we will ignore this subtle point for simplicity until it is time to define vector-Jacobian products later in the chapter.

Done! Let us analyze the algorithm in more detail. First, all the operations we listed can be easily interleaved with the original program, meaning that the space complexity will be roughly proportional to the space complexity of the program we are differentiating.



Light on Memory, Heavy on Arithmetic

On the negative side, the core operation of the algorithm, the update of $\hat{W}_i$, requires a multiplication of two matrices, generically shaped (c'_i, c_i) and (c_i, p_j) , where c_i and c'_i are input and output shapes, and p_j is the shape of $\mathbf{w}_j$. This is an extremely expensive operation. For example, assume that inputs and outputs are both shaped (n, d) , where n is the mini-batch dimension and d represents the input and output features. Then the matrix multiplication will have complexity $O(n^2 d^2 p_j)$, which is quadratic in both the mini-batch size and the feature dimensionality. This can easily become unfeasible, especially for high-dimensional inputs such as images.

We can obtain a better trade-off by noting that the last operation of the algorithm is a simpler matrix-vector product, which is a

consequence of having a scalar output. This is explored in more detail in the next section.

Questions

1. In forward-mode differentiation, what happens to the tangent matrices already stored when a new primitive is applied?
 - a. They are thrown away and computed again at the end
 - b. They are all updated by multiplying them by the input Jacobian of the new primitive
 - c. They are added to the weight Jacobian of the new primitive
 - d. They stay unchanged until the final sum
2. Why is the memory use of forward-mode differentiation modest?
 - a. Because it stores only the final gradient
 - b. Because it never multiplies matrices
 - c. Because its operations can be interleaved with the original program, so its space complexity is roughly that of the program
 - d. Because it uses a vector of ones instead of Jacobians
3. What makes forward-mode differentiation expensive for large inputs such as images?
 - a. Its core update multiplies two matrices, with a cost quadratic in both the mini-batch size and the number of features
 - b. It has to run the program twice

- c. It needs a symbolic engine for every primitive
- d. Its final sum is a matrix-matrix product

Answers

1. **b)** Every time a primitive is applied, its tangent is initialized while all previous tangent matrices are updated by multiplying them by its input Jacobian, which is shared by all of them.
2. **c)** All the operations can be easily interleaved with the original program, so the space complexity is roughly proportional to that of the program being differentiated.
3. **a)** The update of each tangent is a matrix-matrix multiplication with complexity $O(n^2 d^2 p_j)$, quadratic in mini-batch size and feature dimensionality.

Worked examples

Counting tangent updates A program has $l = 4$ primitives. In forward mode, the i -th primitive updates the $i - 1$ tangent matrices already stored. How many tangent updates are made in total, and how many tangents are initialized?

The first primitive finds nothing stored: 0 updates, and it initializes its own tangent.

The second updates the one tangent stored: 1 update.

The third updates two, the fourth updates three.

In total $0 + 1 + 2 + 3 = 6$ updates, each a multiplication of two matrices.

Each of the four primitives initializes one tangent, so 4 tangents are initialized.

Answer: 6 updates and 4 initialized tangents

Practice

1. In the chain rule $\partial_{\mathbf{x}}\mathbf{y} = \partial_{\mathbf{h}}\mathbf{y} \cdot \partial_{\mathbf{x}}\mathbf{h}$, suppose $\mathbf{x}$ has dimension $a = 3$, $\mathbf{h}$ has $b = 4$ and $\mathbf{y}$ has $c = 2$. What are the shapes of the two factors and of the result?

Hint: The book says the product multiplies a $c \times b$ matrix by a $b \times a$ one.

Answer: $\partial_{\mathbf{h}}\mathbf{y}$ is 2×4 and $\partial_{\mathbf{x}}\mathbf{h}$ is 4×3 , so their product $\partial_{\mathbf{x}}\mathbf{y}$ is 2×3 : c by a , as a Jacobian of $\mathbf{y}$ with respect to $\mathbf{x}$ should be.

2. The update in forward mode costs $O(n^2 d^2 p_j)$. If the mini-batch size n doubles while d and p_j stay the same, by what factor does the cost grow? And if both n and d double?

Hint: Look at the powers of n and d .

Answer: Doubling n multiplies the cost by $2^2 = 4$. Doubling both n and d multiplies it by $4 \times 4 = 16$, which is why the method quickly becomes unfeasible for large inputs.

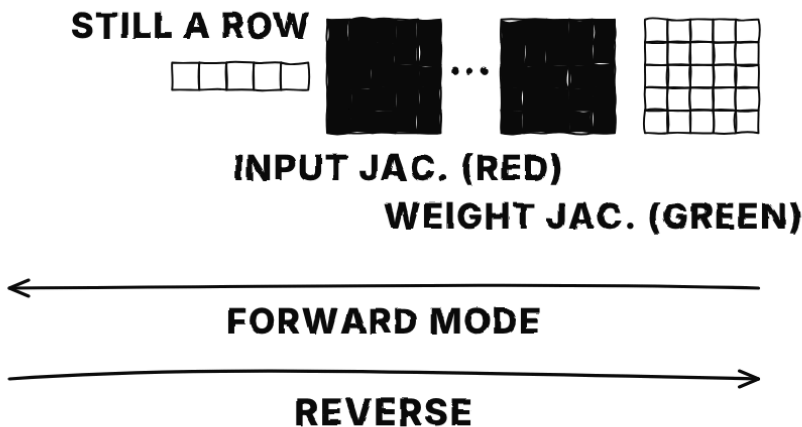
3. Explain in your own words why the book hopes for a better trade-off from the last operation of the algorithm.

Hint: What is special about the output of the program?

Answer: Because the program ends with a sum and its output is a scalar, the last step is a simpler matrix-vector product, the vector of ones transposed times a tangent. A product with a vector is much cheaper than one between two matrices, and the next section exploits this.

Reverse-mode differentiation

To proceed, we unroll the computation of a single gradient term, the one corresponding to the i -th weight matrix. It reads $\nabla_{\mathbf{w}_i} y = \mathbf{1}^T [\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i] \cdots [\partial_{\mathbf{h}_i} \mathbf{h}_{i+1}] [\partial_{\mathbf{w}_i} \mathbf{h}_i]$.

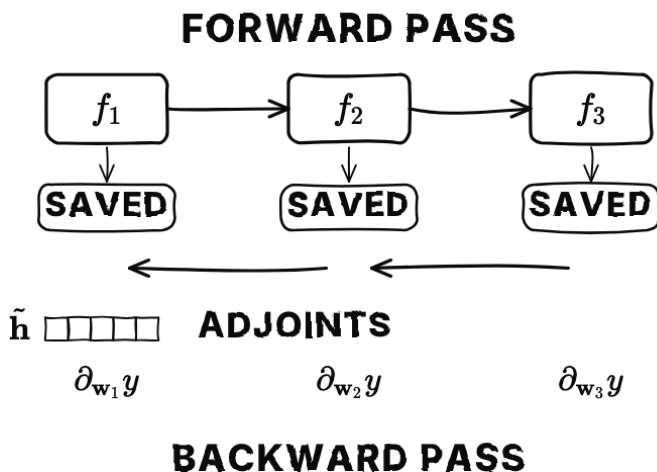


One Gradient as a Row of Products

Remember that, notation apart, this is just a potentially long series of matrix multiplications. It involves a constant term, a vector $\mathbf{1}$ of ones; a series of input Jacobians, which the book prints in red; and a weight Jacobian of the corresponding weight matrix, printed in green. Let us define a shorthand for the red term: $\tilde{\mathbf{h}}_i = \mathbf{1}^T \prod_{j=i+1}^l \partial_{\mathbf{h}_{j-1}} \mathbf{h}_j$.

Because matrix multiplication is associative, we can perform the computations in this product in any order. In F-AD, we proceeded from the right to the left, since that corresponds to the order in which the primitive functions were executed. However, we can do better by noting two interesting aspects. First, the leftmost term in the product is a product between a vector and a matrix, which is a consequence of having a scalar output, and this is computationally better than a product between two matrices. Its output is also another vector. Second, the shorthand term, the product of all input Jacobians from layer i to layer l , can be computed recursively, starting from the last term and iteratively multiplying by the input Jacobians in the reverse order.

We can put these observations together to develop a second approach to automatic differentiation, which we call reverse-mode automatic differentiation, or R-AD, outlined next.



Forward Pass, Then Backward Pass

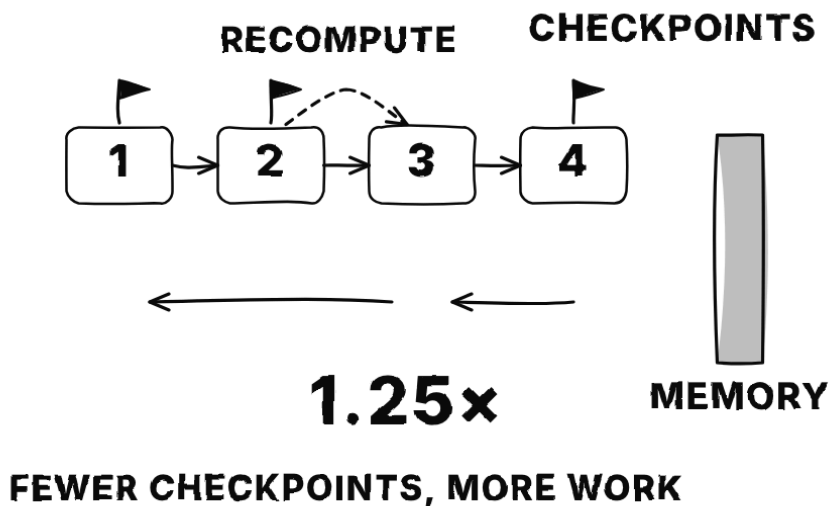
Step one: differently from F-AD, we start by executing the entire program to be differentiated, storing all intermediate outputs. Step two: we initialize a vector $\tilde{\mathbf{h}} = \mathbf{1}^T$, which corresponds to the leftmost term of the product. Step three: moving in reverse order, that is, for an index i ranging over $l, l-1, l-2$, and so on down to 1, we first compute the gradient with respect to the i -th weight matrix as $\partial_{\mathbf{w}_i} y = \tilde{\mathbf{h}} [\partial_{\mathbf{w}_i} \mathbf{h}_i]$, which is the i -th gradient we need. Next, we update our “back-propagated” input Jacobian as $\tilde{\mathbf{h}} \leftarrow \tilde{\mathbf{h}} [\partial_{\mathbf{h}_{i-1}} \mathbf{h}_i]$.

Steps one to three describe a program which is roughly symmetrical to the original program, and which we call the dual, or reverse, program. The terms $\tilde{\mathbf{h}}$ are called the adjoints, and they store, one after the other, all the gradients of the output with respect to the variables

$\mathbf{h}_1, \mathbf{h}_2, \dots, \mathbf{h}_l$ in our program. Compare this with F-AD, where the tangents represented instead the gradients of the variables $\mathbf{h}_i$ with respect to the weights.

In the terminology of neural networks, we sometimes say that the original, or primal, program is a forward pass, not to be confused with forward mode, while the reverse program is a backward pass. Differently from F-AD, in R-AD the full primal program must be executed before the reverse program can be run, and we need specialized mechanisms to store all intermediate outputs in order to “unroll” the computational graph. Different frameworks implement this differently, as outlined next.

Computationally, R-AD is significantly more efficient than F-AD. In particular, both operations in step three of R-AD are vector-matrix products, scaling only linearly in all shape quantities. The trade-off is that executing R-AD requires a large amount of memory, since all intermediate values of the primal program must be stored with a suitable strategy. Specific techniques, such as gradient checkpointing, can be used to improve on this trade-off by increasing the computations and partially reducing the memory requirements. This is done by storing only a few intermediate outputs, called checkpoints, while recomputing the remaining values during the backward pass.



Gradient Checkpointing

Figure 6.1 shows an example of gradient checkpointing in four pictures. In the first, we execute a forward pass, but we only store the outputs of the first, second and fourth blocks, the checkpoints. In the second, the backward pass, drawn with red arrows, stops at the third block, whose activations are not available. In the third, we run a second forward pass, starting from the closest checkpoint, to materialize the activations again. In the fourth, we complete the pass. Compared to a standard backward pass, this requires 1.25 times more computation. In general, the fewer checkpoints are stored, the higher the computational cost of the backward pass.

Questions

1. Why is it allowed to compute the long product for the gradient from left to right instead of from right to left?
 - a. Because matrix multiplication is commutative
 - b. Because matrix multiplication is associative
 - c. Because all the Jacobians are square
 - d. Because the vector of ones can be ignored
2. What makes going from the left, starting with the vector of ones, cheaper?
 - a. Every step is then a product of a vector and a matrix, whose output is again a vector
 - b. The weight Jacobians are no longer needed
 - c. The program no longer needs to be run
 - d. The Jacobians become diagonal
3. What is the price paid for the efficiency of reverse mode?
 - a. It gives only approximate gradients
 - b. It needs a large amount of memory, because all intermediate values of the primal program must be stored
 - c. It can only handle programs with one layer
 - d. Its cost is quadratic in the mini-batch size
4. In gradient checkpointing, what happens if fewer checkpoints are stored?
 - a. Less memory is used and the backward pass costs more computation

- b. More memory is used and the backward pass is faster
- c. Nothing changes, since every value is stored anyway
- d. The gradients become less accurate

Answers

1. **b)** Because matrix multiplication is associative, the computations in the product can be performed in any order.
2. **a)** The leftmost term is a product between a vector and a matrix, a consequence of the scalar output; this is computationally better than a product of two matrices, and its output is another vector.
3. **b)** The full primal program must run first and all its intermediate values must be stored, so reverse mode requires a large amount of memory.
4. **a)** Only a few outputs are stored and the rest are recomputed during the backward pass; the fewer checkpoints are stored, the higher the computational cost of the backward pass.

Worked examples

The cost of checkpointing In the book's figure, checkpointing makes the backward pass need 1.25 times the computation of a standard backward pass. If a standard backward pass on some model takes 8 seconds, how long does it take with this checkpointing?

Checkpointing stores only some outputs, so the missing activations are computed again during the backward pass.

In the book's example this makes the computation 1.25 times that of a standard backward pass.

So the time becomes 1.25×8 seconds.

That is 10 seconds: 2 seconds more, paid in exchange for storing fewer intermediate outputs.

Answer: 10 seconds

Practice

1. A program has $l = 3$ primitives. List, in order, the steps of reverse-mode differentiation: which gradient is computed first, and how the adjoint $\tilde{h}$ changes after each one.

Hint: Start by running the whole program, then set $\tilde{h} = 1^\top$ and move backwards from $i = 3$.

Answer: Run the program and store h_1, h_2, h_3 . Set $\tilde{h} = 1^\top$. For $i = 3$: the gradient for w_3 is $\tilde{h}$ times the Jacobian of h_3 with respect to w_3 , then $\tilde{h}$ is multiplied by the Jacobian of h_3 with respect to h_2 . For $i = 2$: the gradient for w_2 , then $\tilde{h}$ is multiplied by the Jacobian of h_2 with respect to h_1 . For $i = 1$: the gradient for w_1 . The last weight's gradient comes first.

2. Multiplying a row vector of length b by a $b \times a$ matrix takes $b \cdot a$ multiplications; multiplying a $c \times b$ matrix by a $b \times a$ matrix takes $c \cdot b \cdot a$. With $a = b = c = 100$, compare the two.

Hint: Compute both products of the dimensions.

Answer: The vector-matrix product takes $100 \times 100 = 10,000$ multiplications, the matrix-matrix product $100 \times 100 \times 100 = 1,000,000$, a hundred times more. This is why starting from the vector of ones, as reverse mode does, pays off.

3. Explain in your own words the difference between what the tangents of forward mode store and what the adjoints of reverse mode store.

Hint: Gradients of what, with respect to what?

Answer: The tangents of forward mode are the gradients of the intermediate variables h with respect to the weights. The adjoints of reverse mode are the gradients of the output y with respect to the

intermediate variables h_1 , h_2 and so on, stored one after another as the reverse program runs.

Vector-Jacobian products

Practical considerations.

VJP

JACOBIAN

$$\mathbf{v}^\top \times \begin{array}{|c|c|c|c|c|} \hline & & & & \\ \hline & & & & \\ \hline & & & & \\ \hline \end{array} = \begin{array}{|c|c|c|c|c|} \hline & & & & \\ \hline & & & & \\ \hline & & & & \\ \hline \end{array}$$

A ROW AGAIN

$$\begin{array}{|c|c|c|c|c|} \hline & & & & \\ \hline & & & & \\ \hline & & & & \\ \hline \end{array} \times \begin{array}{|c|} \hline \\ \hline \\ \hline \\ \hline \\ \hline \\ \hline \\ \hline \\ \hline \end{array}$$

JVP: FORWARD MODE

A Row Times a Jacobian

Looking at step three of the R-AD algorithm, we can make an interesting observation: the only operation we need is a product between a row vector $\mathbf{v}$ and a Jacobian of f , either the input or the weight Jacobian. We call these two operations the vector-Jacobian products, or VJPs, of f . By contrast, F-AD can be formulated entirely in terms of the transpose of the VJP, called a Jacobian-vector product, or JVP; for a one-dimensional output, the JVP is the directional

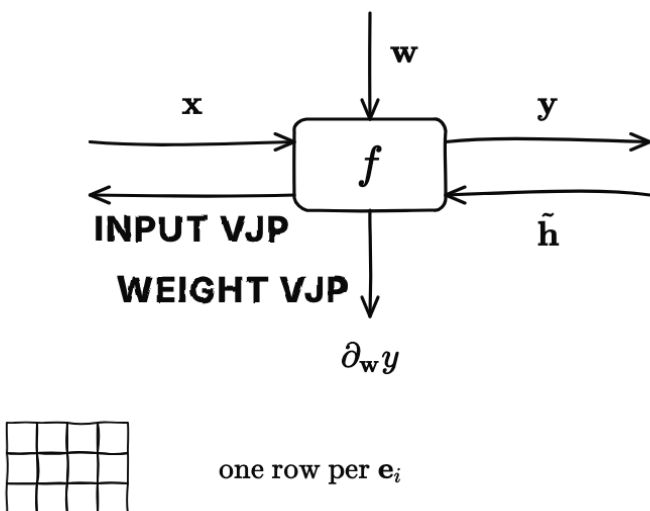
derivative. By analogy, the VJP represents the application of a linear map connected to infinitesimal variations of the output of the function. In the next definition we restore dimensional consistency by adding a transpose to the vector.

Definition: the vector-Jacobian product. Given a function $\mathbf{y} = f(\mathbf{x})$, with $\mathbf{x} \sim (c)$ and $\mathbf{y} \sim (c')$, its VJP is another function, defined as $\text{vjp}_f(\mathbf{v}) = \mathbf{v}^T \partial f(\mathbf{x})$, where $\mathbf{v} \sim (c')$. If f has multiple parameters, $f(\mathbf{x}_1, \dots, \mathbf{x}_n)$, we can define n individual VJPs, one for each of $\mathbf{x}_1, \dots, \mathbf{x}_n$.

In particular, in our case we can define two types of VJPs, corresponding to the partial derivatives of the primitive with respect to the input and to the weight arguments. The input VJP is $\text{vjp}_{f,\mathbf{x}}(\mathbf{v}) = \mathbf{v}^T \partial_{\mathbf{x}} f(\mathbf{x}, \mathbf{w})$. The weight VJP is $\text{vjp}_{f,\mathbf{w}}(\mathbf{v}) = \mathbf{v}^T \partial_{\mathbf{w}} f(\mathbf{x}, \mathbf{w})$.

We can now rewrite the two operations in step three of the R-AD algorithm as two VJP calls of the primitive function with the adjoint values, ignoring the index i for readability: the adjoint times the weight VJP, and the adjoint times the input VJP. The first gives the gradient, $\partial_{\mathbf{w}} y = \text{vjp}_{f,\mathbf{w}}(\tilde{\mathbf{h}})$. The second updates the adjoint, $\tilde{\mathbf{h}} \leftarrow \text{vjp}_{f,\mathbf{h}}(\tilde{\mathbf{h}})$.

Hence we can implement an entire automatic differentiation system by first choosing a set of primitive operations, and then augmenting them with the corresponding VJPs, without having to materialize the Jacobians in memory at any point. This is shown schematically in Figure 6.2: to perform R-AD, primitives must be augmented with two VJP operations, the input VJP and the weight VJP, and one call of each is sufficient to perform the backward pass through the primitive.



A Primitive With Two VJPs

In fact, we can recover the Jacobians by repeatedly calling the VJPs with the basis vectors $\mathbf{e}_1, \dots, \mathbf{e}_n$, generating them one row at a time. For the input Jacobian, for example, the rows of $\partial_{\mathbf{x}} f(\mathbf{x}, \mathbf{w})$ are $\text{vjp_f, x}(\mathbf{e}_1)$, $\text{vjp_f, x}(\mathbf{e}_2)$, and so on down to $\text{vjp_f, x}(\mathbf{e}_n)$.

To understand why this reformulation can be convenient, let us look at the VJPs of a fully-connected layer, which is composed of linear projections and elementwise non-linearities. First, consider a simple linear projection with no bias, $f(\mathbf{x}, \mathbf{W}) = \mathbf{W}\mathbf{x}$. The input Jacobian here is simply $\mathbf{W}$, but the weight Jacobian is a rank-3 tensor. By comparison, the input VJP has no special structure: $\text{vjp_f, x}(\mathbf{v}) = \mathbf{v}^T \mathbf{W}^T = [\mathbf{W}\mathbf{v}]^T$. The weight VJP, instead, turns out to be a simple outer product, which avoids rank-3 tensors completely: $\text{vjp_f, w}(\mathbf{v}) = \mathbf{v}\mathbf{x}^T$.

The diagram illustrates the outer product of two vectors. On the left, a vertical vector $\mathbf{v}$ (represented as a 3x1 grid with the middle cell shaded) is multiplied by a horizontal vector $\mathbf{x}^\top$ (represented as a 1x4 grid with the third cell shaded). The result is a 3x4 grid representing the gradient $\partial \mathbf{W}$, where the element at row i and column j is $v_i x_j$. The shaded cells in the input vectors correspond to the shaded cell in the resulting matrix.

The diagram shows an elementwise scaling operation. A horizontal vector $\mathbf{v}$ (1x4 grid) is multiplied elementwise (indicated by a circle with a dot) by the derivative of the activation function φ' (1x4 grid). The result is a scaled vector (1x4 grid), with the word **SCALE** written below it.

Outer Product and Scaling

To work out this VJP, we can write $\mathbf{y} = \mathbf{v}^\top \mathbf{W} \mathbf{x} = \sum_i \sum_j W_{ij} v_i x_j$. From this we immediately get $\partial \mathbf{y} / \partial W_{ij} = v_i x_j$, which is the elementwise definition of the outer product.

Hence, every time we apply a linear projection in the forward pass, we modify the back-propagated gradients by the transpose of its weights, and we perform an outer product to compute the gradient of $\mathbf{W}$.

Consider now an elementwise activation function with no trainable parameters, for example the ReLU: $f(\mathbf{x}) = \varphi(\mathbf{x})$, with an empty set of weights. Because we have no trainable parameters, we need only consider the input VJP. The gradient is a diagonal matrix having as elements the derivatives of φ : its i -th diagonal entry is $\varphi'(x_i)$. The

input VJP is a multiplication of a diagonal matrix by a vector, which is equivalent to a Hadamard product, that is, a scaling operation: $\text{vjp_x}(f, \mathbf{v}) = \mathbf{v} \odot \phi'(\mathbf{x})$. Interestingly, in this case too we can compute the VJP without having to materialize the full diagonal matrix.

Questions

1. Why are vector-Jacobian products all that reverse-mode differentiation needs from a primitive?
 - a. Because step three of the algorithm only ever multiplies a row vector, the adjoint, by an input or a weight Jacobian
 - b. Because the Jacobians of primitives are always vectors
 - c. Because the forward pass computes the Jacobians anyway
 - d. Because a VJP is cheaper to store than the input
2. What is gained by giving each primitive its two VJPs instead of its Jacobians?
 - a. The gradients become exact instead of approximate
 - b. The forward pass no longer has to be run
 - c. The Jacobians never have to be materialized in memory
 - d. The primitives no longer need weights
3. For a linear projection $f(\mathbf{x}, \mathbf{W}) = \mathbf{W}\mathbf{x}$, why is the weight VJP convenient?
 - a. It is zero
 - b. It is a simple outer product, $\mathbf{v} \mathbf{x}^T$, which avoids the rank-3 weight Jacobian completely

- c. It equals the input Jacobian
 - d. It needs the diagonal matrix of derivatives
4. What does the input VJP of an elementwise activation such as the ReLU amount to?
- a. A multiplication by the transpose of the weights
 - b. An outer product
 - c. A scaling of v , element by element, by the derivatives $\phi'(x)$
 - d. A sum over all elements of v

Answers

1. **a)** The only operation step three needs is a product between a row vector and a Jacobian of f , the input or the weight Jacobian; these are the two VJPs of f .
2. **c)** An entire automatic differentiation system can be built from primitives augmented with their VJPs, without having to materialize the Jacobians in memory at any point.
3. **b)** The weight Jacobian is a rank-3 tensor, but the weight VJP turns out to be the outer product $v x^T$, which avoids rank-3 tensors completely.
4. **c)** Its gradient is a diagonal matrix of the derivatives of ϕ , and multiplying a vector by it is a Hadamard product, $v \odot \phi'(x)$, a scaling that needs no diagonal matrix in memory.

Worked examples

The weight VJP as an outer product A linear projection $f(x, W) = Wx$ takes $x = (3, 4, 5)$ to two outputs, so W is 2×3 . The adjoint arriving from above is $v = (1, 2)$. Compute the weight VJP, that is, the gradient for W .

The book shows that $\partial y / \partial W_{ij} = v_i x_j$, so the weight VJP is the outer product $v x^T$.

Row 1 uses $v_1 = 1$: $(1 \times 3, 1 \times 4, 1 \times 5) = (3, 4, 5)$.

Row 2 uses $v_2 = 2$: $(2 \times 3, 2 \times 4, 2 \times 5) = (6, 8, 10)$.

The result has the same 2×3 shape as W , and no rank-3 tensor was ever built.

Answer: The 2×3 matrix with rows $(3, 4, 5)$ and $(6, 8, 10)$

Practice

1. A ReLU receives $x = (1, -2, 0.5)$; its derivative is 1 where the input is positive and 0 where it is negative. The adjoint is $v = (2, -1, 3)$. Compute the input VJP $v \odot \phi'(x)$.

Hint: First write $\phi'(x)$ element by element, then multiply element by element.

Answer: $\phi'(x) = (1, 0, 1)$, so $v \odot \phi'(x) = (2, 0, 3)$. The middle component is switched off because its input was negative.

2. A primitive has 4 outputs. How many calls of its input VJP do you need to rebuild its full input Jacobian, and with which vectors?

Hint: Each call with a basis vector gives one row.

Answer: Four calls, with the basis vectors e_1, e_2, e_3 and e_4 ; each call returns one row of the input Jacobian.

3. Explain in your own words what happens to the back-propagated gradient, and what is computed for the weights, each time the forward pass contains a linear projection.

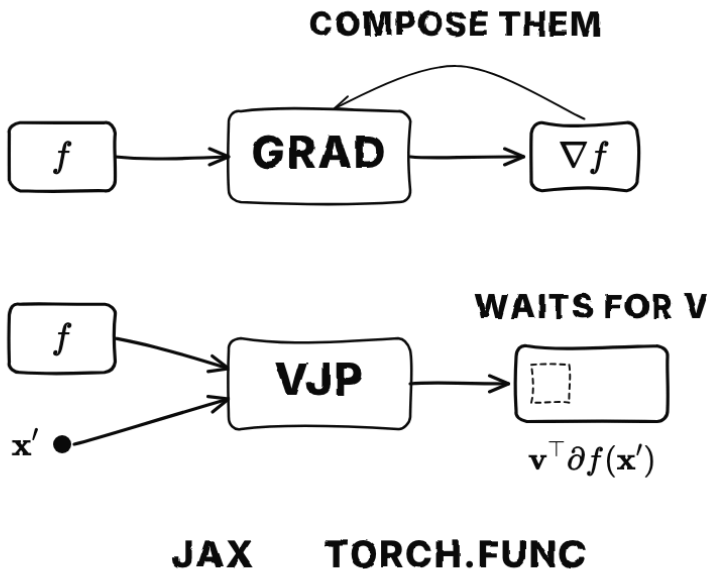
Hint: There are two VJPs: one for the input, one for the weights.

Answer: The adjoint passing backwards is multiplied by the transpose of the layer's weights, through the input VJP. At the same time, the gradient

of W is the outer product of the adjoint and the layer's input, through the weight VJP.

Implementing a reverse-mode system

There are many ways to implement the R-AD system, ranging from Wengert lists, as done in TensorFlow, to source-to-source code transformations. Here we briefly discuss some common implementations in existing frameworks.



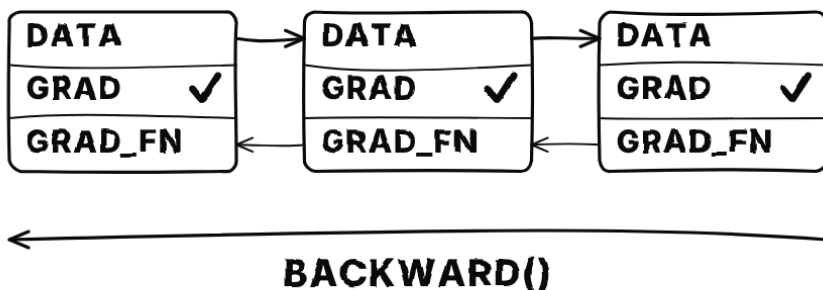
Functions That Make Functions

First, describing primitives as functions with two arguments, $f(\mathbf{x}, \mathbf{w})$, aligns with functional frameworks such as JAX, where everything is a function. Consider a function $f(\mathbf{x})$ with a c -dimensional input and a c' -dimensional output. From this point of view, a VJP can be

implemented as a higher-order function with the signature $(\mathbb{R}^c \rightarrow \mathbb{R}^{c'}) \rightarrow \mathbb{R}^c \rightarrow (\mathbb{R}^{c'} \rightarrow \mathbb{R}^c)$. That is, given a function f and an input $\mathbf{x}'$, a VJP returns another function, which can be applied to a c' -dimensional vector $\mathbf{v}$ to return $\mathbf{v}^\top \partial f(\mathbf{x}')$.

Similarly, the gradient of a one-dimensional function can be implemented as another higher-order function, with the signature $(\mathbb{R}^c \rightarrow \mathbb{R}) \rightarrow (\mathbb{R}^c \rightarrow \mathbb{R}^c)$. It takes as input the function $f(\mathbf{x})$ and returns another function, which computes $\nabla f(\mathbf{x})$. In JAX, these ideas are implemented in functions such as `jax.grad` and `jax.vjp`, and they are replicated in PyTorch in the `torch.func` module. In the book's Box 6.2, for example, a sum-of-squares function is defined, `func.grad` turns it into its gradient function, and applied to a random vector of ten numbers it returns a gradient of size ten. The `torch.func` interface replicates the JAX interface, and in practice the function can be traced, for example with `torch.compile`, to generate an optimized computational graph. Many operations, such as computing a Hessian, can be achieved by smartly composing JVPs and VJP according to their signatures.

As we mentioned, in practice our models are implemented as compositions of objects whose parameters are encapsulated as properties. One possibility is to “purify” the object, turning it into a pure function. In PyTorch, for example, one extracts the parameters of the model into a dictionary, and then calls the model's forward function through `torch.func.functional_call`, passing the model, the parameters and the input.



MICROGRAD

Tensors That Remember Their Origin

More generally, frameworks like PyTorch are augmented with techniques to handle this scenario directly, without introducing intermediate operations. In PyTorch, for example, tensor objects are augmented with information about the operation that generated them. Whenever a `backward()` call is requested on a scalar value, these properties are used to traverse the computational graph in reverse, storing the corresponding gradients inside the tensors that require them.

Figure 6.3 shows this in two halves. On the left, a PyTorch tensor carries information about its gradient, empty at initialization, and about the operation that created it. On the right, during a backward pass, the `grad_fn` property is used to traverse the computational

graph in reverse, and gradients are stored inside the tensor's `grad` property whenever `requires_grad` is explicitly set to true, so as not to consume unnecessary memory.

This is just a high-level overview of how these systems are implemented in practice, and we are leaving out many details, for which we refer to the official documentation. The author warmly suggests trying to implement a reverse-mode system from scratch; many didactic implementations can be found online, such as Andrej Karpathy's `micrograd`.

Questions

1. In a functional framework such as JAX, what does the gradient operation take and return?
 - a. It takes a number and returns a number
 - b. It takes a function f and returns another function, which computes the gradient of f at any input
 - c. It takes a tensor and stores the gradient inside it
 - d. It takes a list of operations and returns their Jacobians
2. What does a VJP return when it is given a function f and an input x' ?
 - a. The full Jacobian of f at x'
 - b. The value $f(x')$
 - c. Another function, which applied to a vector v returns v^T times the Jacobian of f at x'
 - d. A list of the parameters of f

3. How does PyTorch know how to go backwards through the computational graph when `backward()` is called?
- a. Each tensor carries information about the operation that created it, which is followed in reverse
 - b. It differentiates the source code symbolically
 - c. It reruns the forward pass with small changes to every input
 - d. It asks the user to list the operations

Answers

1. **b)** The gradient is a higher-order function: it takes the function $f(x)$ as input and returns another function that computes $\nabla f(x)$.
2. **c)** Given f and x' , a VJP returns another function that can be applied to a c' -dimensional vector v to return $v^T \partial f(x')$.
3. **a)** Tensor objects are augmented with information about the operation that generated them, and on a `backward()` call these properties are used to traverse the graph in reverse.

Worked examples

The gradient of the sum of squares In the book's Box 6.2 the function is $f(x) = \text{sum of the squares of the elements of } x$. Its gradient is $2x$. What does the gradient function return for $x = (1, -2, 3)$, and what is its shape compared to x ?

Each element x_k appears in f only as x_k^2 , whose derivative is $2x_k$.

So the gradient is $(2 \times 1, 2 \times (-2), 2 \times 3) = (2, -4, 6)$.

The gradient has one entry per input element, so its shape is that of x , just as in the box a vector of size 10 gives a gradient of size 10.

Answer: $(2, -4, 6)$, with the same shape as x

Practice

1. Read the signature of the gradient, $(\mathbb{R}^c \rightarrow \mathbb{R}) \rightarrow (\mathbb{R}^c \rightarrow \mathbb{R}^c)$, in your own words. Why must its input be a function with a one-dimensional output?

Hint: Look at what goes in on the left and what comes out on the right.

Answer: It takes a function from c numbers to one number, and returns a function from c numbers to c numbers. The gradient is defined for a scalar output: for each of the c inputs it gives one partial derivative, so the result has c entries.

2. Explain why a model written as an object with parameters must be “purified” before it can be used with functional tools such as `func.grad`.

Hint: Functional tools differentiate with respect to arguments of a function.

Answer: Functional tools treat everything as a function and differentiate with respect to its arguments, as in $f(x, w)$. In an object the parameters are hidden inside as properties, so they are first extracted and then passed explicitly through a functional call, which makes the model a pure function of its parameters and input.

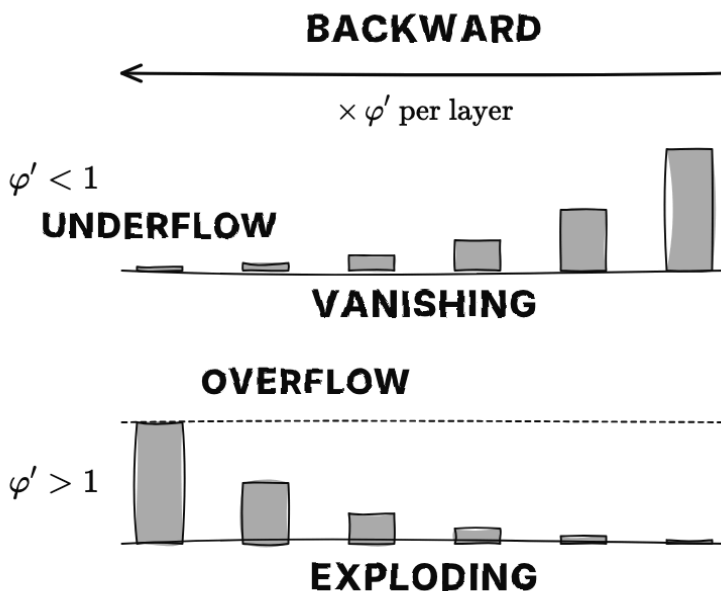
3. In PyTorch, why are gradients stored only in the tensors whose `requires_grad` is set to true?

Hint: Think about memory.

Answer: Storing a gradient inside every tensor would consume memory needlessly; PyTorch stores gradients only where they are explicitly required, typically the parameters being trained.

Choosing an activation function

Coincidentally, we can now motivate why the ReLU is a good choice of activation function. A close look at the input VJP of an activation function tells us that every time we add an activation function to our model, the adjoints in the backward pass are scaled by a factor of $\phi'(\mathbf{x})$. For models with many layers, this can give rise to two pathological behaviours. If $\phi' < 1$ everywhere, there is the risk of the gradient shrinking to zero exponentially fast in the number of layers. This is called the vanishing gradient problem. Conversely, if $\phi' > 1$ everywhere, the opposite problem appears, with the gradients growing towards infinity exponentially fast in the number of layers. This is called the exploding gradient problem.



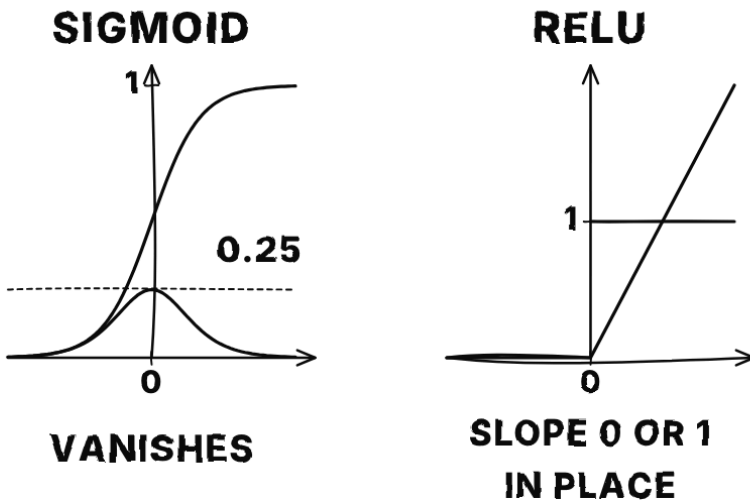
Vanishing and Exploding Gradients

These are serious problems in practice, because libraries represent floating-point numbers with limited precision, typically 32 bits or lower, meaning that underflows or overflows can appear quickly as the number of layers increases.

Surprisingly, a stack of linear layers implemented in floating-point precision is not fully linear, because of small discontinuities at machine precision! This is generally not an issue, but it can be exploited to train fully linear deep neural networks.

As an example of how vanishing gradients can appear, consider the sigmoid function, $\sigma(s)$. We already mentioned that this was a common activation function in the past, because it is a soft approximation to

the step function. We also know that $\sigma'(s) = \sigma(s)(1 - \sigma(s))$. Combined with the fact that $\sigma(s)$ lies between 0 and 1, we obtain that $\sigma'(s)$ lies between 0 and 0.25. Hence the sigmoid is a prime candidate for vanishing gradient issues. Figure 6.4 shows, on the left, the sigmoid and its derivative, and on the right, the ReLU and its derivative.



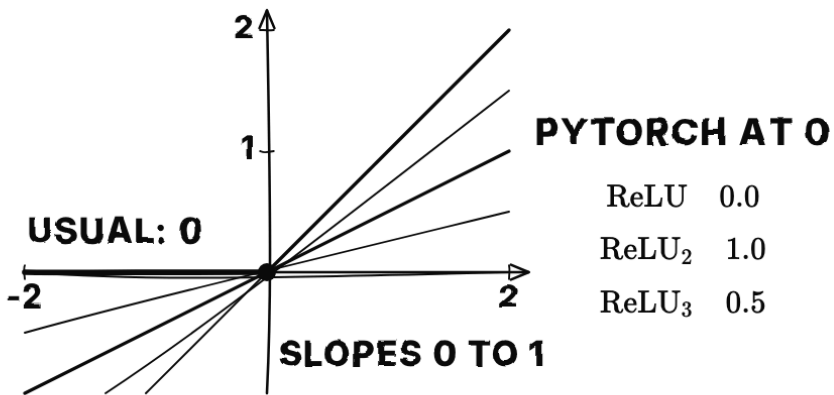
Sigmoid Versus ReLU Slopes

Designing an activation function that never exhibits vanishing or exploding gradients is not trivial, since the only function having $\phi'(s) = 1$ everywhere is the identity function. We then need a function which is “linear enough” to avoid gradient issues, but “non-linear” enough to separate the linear layers. The ReLU ends up being a good candidate, since its derivative is 0 for s less than 0, and 1 for s greater than 0. The gradient is either zeroed out, inducing sparsity in the

computation, or multiplied by 1, avoiding scaling issues; this is shown on the right of Figure 6.4.

As a side note, the gradient of the ReLU is identical whether we compute it from the input of the ReLU layer or from its output, since we are only masking the negative values while keeping the positive values untouched. Hence another benefit of using the ReLU as activation function is that we can save a little memory when performing R-AD, by overwriting the layer's input in the forward pass without impacting the correctness of the AD procedure. This is done in PyTorch, for example, by setting the in-place parameter.

Subdifferentiability and automatic differentiation.



Many Slopes at the Corner

There is a small detail we avoided discussing until now: the ReLU is not differentiable at 0, which makes the overall network non-smooth. What happens in this case? The “pragmatic” answer is that, by minimizing with stochastic gradient descent from a random, non-zero initialization, the probability of ending up exactly at $s = 0$ is practically null, while the gradient is defined at $\text{ReLU}(\varepsilon)$ for any epsilon different from zero.

For a more technical answer, we can introduce the concept of the subgradient of a function. Definition: given a convex function $f(x)$, a subgradient at x is a number z such that, for all y , $f(y) \geq f(x) + z(y - x)$.

Note the similarity with the definition of convexity: a subgradient is the slope of a line “tangent” to $f(x)$, such that the entire function f lies above it. If f is differentiable at x , then only one such line exists, and its slope is the derivative of f at x . At a non-smooth point, multiple subgradients exist, and they form a set called the subdifferential of f at x : the set of all z that are subgradients of $f(x)$.

With this definition in hand, we can complete our analysis of the gradient of the ReLU by replacing the gradient with its subdifferential at 0. For s less than 0, the subdifferential contains only 0; for s greater than 0, it contains only 1; and at $s = 0$ it is the whole interval from 0 to 1. Hence any value between 0 and 1 is a valid subgradient at 0, with most implementations in practice choosing $\text{ReLU}'(0) = 0$. Selecting subgradients at every step of an iterative descent procedure is called subgradient descent.

In fact, the situation is even more tricky, because the subgradient need not be defined for non-convex functions. In that case, one can resort to generalizations that relax the previous definition to a local neighbourhood of x , such as the Clarke subdifferential.

Subdifferentiability can also create problems in AD, where different implementations of the same function can provide different, and possibly invalid, subgradients, and more refined versions of the chain rule must be considered for a formal proof. Consider this example: define two functions, $\text{ReLU}_2(s) = \text{ReLU}(-s) + s$ and $\text{ReLU}_3(s) = 0.5(\text{ReLU}(s) + \text{ReLU}_2(s))$. Both are equivalent to the ReLU, but in PyTorch a backward pass at 0 returns 0.0 for the ReLU, 1.0 for ReLU_2 , and 0.5 for ReLU_3 .

From theory to practice.

If you followed the exercises in Chapter 5, you already saw an application of R-AD in both PyTorch and JAX, and this chapter, especially the section on implementing a reverse-mode system, should have clarified how they are implemented.

It is a good idea to try to re-implement a simple R-AD system, similar to the one of PyTorch. For example, focusing on scalar-valued quantities, the micrograd repository is a very good didactic implementation. The only detail we do not cover is that, once you move to a general acyclic graph, ordering the variables of the computational graph before the backward pass is essential to avoid creating wrong backpropagation paths. In micrograd, this is achieved with an inexpensive topological sorting of the variables.

It is also interesting to try to implement a new primitive, in the sense used in this chapter, in PyTorch, which requires specifying its forward pass along with its VJPs. One example can be one of the trainable activation functions from Chapter 5. This is a didactic exercise, in the sense that it can be implemented equivalently by subclassing `nn.Module` and letting the AD engine of PyTorch work out the backward pass.

All these steps can also be replicated in JAX. Implement a didactic version of JAX with autodidax. Write out a new primitive by implementing the corresponding VJP. And read the JAX Autodiff Cookbook to discover advanced use cases for the automatic differentiation engine, such as higher-order derivatives, Hessians, and more.

Questions

1. Why can an activation function whose derivative is below 1 everywhere make training of a deep model fail?
 - a. Because the adjoints are scaled by ϕ' at every layer, so the gradient can shrink towards zero exponentially in the number of layers
 - b. Because the forward pass becomes linear
 - c. Because the weights grow towards infinity
 - d. Because the subgradient is no longer defined
2. Why is the sigmoid a prime candidate for vanishing gradients?
 - a. Because its derivative can be larger than 1
 - b. Because its derivative lies between 0 and 0.25
 - c. Because it is not differentiable at 0
 - d. Because it is the identity function
3. Why can the ReLU's input be overwritten by its output during the forward pass without breaking reverse mode?
 - a. Because the ReLU has no backward pass

- b. Because its gradient is the same whether computed from the input or from the output, since only negative values are masked
 - c. Because PyTorch recomputes all inputs anyway
 - d. Because the ReLU's output is always zero
4. What is the subdifferential of the ReLU at 0?
- a. Only the value 0
 - b. Only the value 1
 - c. The whole interval from 0 to 1
 - d. It is empty

Answers

- 1. a)** Every activation scales the adjoints in the backward pass by $\phi'(x)$; if $\phi' < 1$ everywhere, the gradient risks shrinking to zero exponentially fast in the number of layers: the vanishing gradient problem.
- 2. b)** Since $\sigma'(s) = \sigma(s)(1 - \sigma(s))$ and $\sigma(s)$ lies between 0 and 1, the derivative lies between 0 and 0.25, so every sigmoid layer shrinks the gradient.
- 3. b)** The ReLU only masks the negative values and leaves the positive ones untouched, so its gradient is identical whether taken from its input or its output; this saves a little memory.
- 4. c)** At $s = 0$ every slope between 0 and 1 gives a line below the ReLU, so the subdifferential is the interval from 0 to 1; most implementations pick 0.

Worked examples

Ten sigmoid layers In a model with 10 sigmoid layers, each layer scales the adjoint by at most 0.25, the largest value of σ' . At most how

large can the factor be by which the gradient has shrunk after passing back through all 10 layers?

The book shows that $\sigma'(s)$ lies between 0 and 0.25.

Each layer multiplies the back-propagated adjoint by σ' , so by at most 0.25.

Ten layers multiply by at most 0.25^{10} .

$0.25^{10} = 1 / 4^{10} = 1 / 1,048,576$, about 0.00000095.

Even in the best case the gradient is about a millionth of its size: a vanishing gradient.

Answer: At most about 0.00000095, one part in 1,048,576

Practice

1. At some input the sigmoid gives $\sigma(s) = 0.9$. Compute $\sigma'(s)$ from the formula $\sigma'(s) = \sigma(s)(1 - \sigma(s))$.

Hint: Multiply $\sigma(s)$ by one minus $\sigma(s)$.

Answer: $\sigma'(s) = 0.9 \times 0.1 = 0.09$, well below the maximum of 0.25.

2. Using the definition $f(y) \geq f(x) + z(y - x)$ for all y , check whether $z = 0.3$ and $z = 1.2$ are subgradients of the ReLU at $x = 0$.

Hint: At $x = 0$ the condition reads $\text{ReLU}(y) \geq z \cdot y$; try a positive and a negative y .

Answer: For $z = 0.3$: if $y > 0$, $y \geq 0.3y$ holds; if $y < 0$, $0 \geq 0.3y$ holds; so 0.3 is a subgradient. For $z = 1.2$: with $y = 1$, $1 \geq 1.2$ is false, so 1.2 is not. This agrees with the subdifferential being the interval from 0 to 1.

3. The book's $\text{ReLU}_2(s) = \text{ReLU}(-s) + s$ equals the ReLU. Assuming PyTorch takes the ReLU's derivative at 0 to be 0, explain why a backward pass at 0 returns 1.0 for ReLU_2 and 0.5 for $\text{ReLU}_3(s) = 0.5(\text{ReLU}(s) + \text{ReLU}_2(s))$.

Hint: Differentiate each term the way the AD engine does, using $\text{ReLU}'(0) = 0$.

Answer: For ReLU_2 , the engine gets $-\text{ReLU}'(0) + 1 = 0 + 1 = 1$. For ReLU_3 it gets $0.5 \times (\text{ReLU}'(0) + \text{ReLU}_2'(0)) = 0.5 \times (0 + 1) = 0.5$. The three functions are equal, yet the engine returns three different subgradients.