

Backpropagation

Last modified: 2026-10-25

```
# The Python chunks on this page import numpy, pandas, matplotlib and scikit-learn; reticulate provisions
# The Julia chunks run through JuliaCall.
library(reticulate)
py_require(c("numpy", "pandas", "matplotlib", "scikit-learn"))
```

i Source

Built from the training and back-propagation half of Brian Hutchinson’s Fall 2025 lecture notes, which the neural networks chapter left as an outtake. The source states the notation — pre-activations $\tilde{a}$ and $\tilde{b}$, post-activations $\tilde{h}$ and $\tilde{y}$ — and stops there, so we wrote the derivations, the cost accounting and the depth material here. The section on gradients with respect to the input comes from Logan Sizemore’s Spring 2025 lecture on backpropagation. We converted the sections on exploding gradients, choosing the hidden non-linearity and residual connections, and the remark on shattered gradients, from the “Training deep networks” material the neural networks chapter keeps as an outtake, and added their measurements and He initialization.

We computed every number in the solutions rather than quoting it. We also checked the three gradient formulas — single example, K layers, and minibatched — against central differences on random networks, and they agreed to about one part in 10^9 .

1 Why this needs an algorithm

The neural networks chapter built a model and gradient descent built a way to fit one. Putting them together needs one thing neither chapter supplied: the gradient of a network’s loss with respect to its weights.

This looks like a detail. A gradient is a list of partial derivatives, each partial derivative is a limit, and we can approximate a limit by moving the parameter a little and dividing. We need no new mathematics at all.

The trouble is arithmetic rather than mathematics, so we first measure how large it is before looking for a way to avoid it.

Exercise 1.1 (Price the obvious way of getting a gradient). You already know one way to compute a derivative without doing any calculus: move a parameter a little, see how much the loss changes, and divide. For a single weight w , the *central difference* estimate is

$$\frac{\partial \mathcal{L}}{\partial w} \approx \frac{\mathcal{L}(w + \varepsilon) - \mathcal{L}(w - \varepsilon)}{2\varepsilon}.$$

This needs no derivation, works for any loss, and is easy to get right. Before building anything more clever, find out what it would cost.

Take the network from the neural networks chapter: $p = 784$ inputs^a, $L = 100$ hidden units, $C = 10$ outputs, so W is 785×100 and U is 101×10 .

How many parameters does the network have, and how many forward passes does the central-difference estimate need to produce the whole gradient?

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Solution

Solution 1.1. W holds $785 \times 100 = 78,500$ parameters and U holds $101 \times 10 = 1,010$, so $P = 79,510$. Almost 99% of them are in W .

Each parameter needs two losses, at $w + \varepsilon$ and $w - \varepsilon$, and each loss is a full forward pass. That is $2P = 159,020$ forward passes for one gradient. (A one-sided difference halves this to $P + 1$ and is considerably less accurate; Solution 1.4 says why.)

Exercise 1.2 (Price the obvious way of getting a gradient (part 2)). Use the setup of Exercise 1.1. Count the multiply-accumulate operations in one forward pass. (Count only the matrix products^a; the activation functions are one operation per unit and do not change the order of magnitude.) Multiply through: what does one numerical gradient cost, for one training example?

^a<https://morrison-lab.github.io/mds/linear-algebra-matrices.html#def-matrix-mult>

Solution

Solution 1.2. The forward pass is two matrix products^a: $W^\top \tilde{x}$ costs $785 \times 100 = 78,500$ multiply-accumulates and $U^\top [1; \tilde{h}]$ costs $101 \times 10 = 1,010$, for 79,510 in total — one per parameter, which is not a coincidence. Every weight is used exactly once in a forward pass.

So one numerical gradient, for one training example, costs

$$159,020 \times 79,510 \approx 1.26 \times 10^{10} \text{ multiply-accumulates.}$$

The cost is quadratic in the number of parameters, which is the real problem: doubling the width of the hidden layer multiplies the cost by four.

^a<https://morrison-lab.github.io/mds/linear-algebra-matrices.html#def-matrix-mult>

Exercise 1.3 (Price the obvious way of getting a gradient (part 3)). Use the setup of Exercise 1.1. MNIST has 60,000 training images. What does one epoch of full-batch gradient descent^a cost this way? Compare it with a method that computes the whole gradient in the time of about three forward passes.

^a<https://morrison-lab.github.io/mds/optimization.html#def-gradient-descent>

Solution

Solution 1.3. Multiply by 60,000:

$$60,000 \times 1.26 \times 10^{10} \approx 7.6 \times 10^{14}$$

multiply-accumulates for a single step. A method costing three forward passes needs $60,000 \times 3 \times 79,510 \approx 1.4 \times 10^{10}$, a factor of about 53,000 less. Put differently: one step of the numerical method costs what fifty thousand steps of the other one costs. Training is thousands of steps, so this is not a constant^a factor to optimize later — it decides whether we can fit the model at all.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

Exercise 1.4 (Price the obvious way of getting a gradient (part 4)). Use the setup of Exercise 1.1. The method has an accuracy problem as well as a speed problem. Taking ε too large leaves a truncation error, and taking it too small leaves a rounding error, because the two losses being subtracted agree in their leading digits. Which way does each error move as ε shrinks, and roughly where is the best compromise in double precision?

Solution

Solution 1.4. Write $\mathcal{L}$ as a smooth function of the one weight. Taylor expansion gives

$$\frac{\mathcal{L}(w + \varepsilon) - \mathcal{L}(w - \varepsilon)}{2\varepsilon} = \mathcal{L}'(w) + \frac{\varepsilon^2}{6} \mathcal{L}'''(w) + \dots,$$

so the truncation error falls like ε^2 . Against that, the computer evaluates both losses to a relative accuracy of about $\varepsilon_{\text{mach}} \approx 2.2 \times 10^{-16}$, and the subtraction cancels their leading digits, so the rounding error in the quotient grows like $\varepsilon_{\text{mach}}/\varepsilon$.

The total is smallest where the two are comparable, at $\varepsilon \sim \varepsilon_{\text{mach}}^{1/3} \approx 6 \times 10^{-6}$, leaving an error of about $\varepsilon_{\text{mach}}^{2/3} \approx 4 \times 10^{-11}$. A one-sided difference has truncation error $O(\varepsilon)$ instead, so its best compromise is at $\varepsilon \sim \sqrt{\varepsilon_{\text{mach}}} \approx 1.5 \times 10^{-8}$, where the truncation and rounding terms are both that size, leaving an error worse by a factor of $\sqrt{\varepsilon_{\text{mach}}}/\varepsilon_{\text{mach}}^{2/3} \approx 400$ for half the work.

Neither is catastrophic on its own. You pay 10^{10} operations for an *approximate* gradient when an exact one is available for 10^5 .

Exercise 1.5 (Price the obvious way of getting a gradient (part 5)). Use the setup of Exercise 1.1. The gradient is a list of P numbers, so no method can produce it in less than P operations. Say what the best imaginable algorithm^a would cost here, in units of forward passes, and what that target implies about how much work each parameter may receive.

^a<https://morrison-lab.github.io/mds/expressions-algorithms.html#def-algorithm>

Solution

Solution 1.5. The output has P numbers in it, so P operations is a lower bound, and one forward pass already costs P here. The best imaginable algorithm^a therefore runs in a small constant^b number of forward passes — and the constant has to be small enough that each parameter receives only $O(1)$ work per training example.

That is exactly what back-propagation achieves, and the rest of this chapter builds it. Solution 1.2 is where to look for the reason: a forward pass touches each weight once, so a single pass already carries past each weight everything that is local to it. It carries nothing about the loss, which sits at the far end of the network. The next few exercises ask what has to come back the other way, and how little of it there turns out to be.

^a<https://morrison-lab.github.io/mds/expressions-algorithms.html#def-algorithm>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

i Watch

- Sanderson explains backpropagation with pictures and little algebra, from 3:07^a [1, Ch. 3].
- Starmer explains the main ideas of backpropagation^b [2].

^a<https://www.youtube.com/watch?v=Ilg3gGewQ5U&t=187s>

^b<https://www.youtube.com/watch?v=IN2XmBhILt4>

i Watch

Hutchinson covers the idea of backpropagation^a (15 min) [3]. The video site requires a login, which enrolled students find in the syllabus.

^ahttps://facultyweb.cs.wvu.edu/~hutchib2/video_lectures/data371/#intro_to_backprop

i Read

GeeksforGeeks explains backpropagation^a [4].

^a<https://www.geeksforgeeks.org/machine-learning/backpropagation-in-neural-network/>

2 Naming the pieces

The obvious method fails because it treats the network as a function it can only evaluate, and evaluates it 159,020 separate times. Everything that follows comes from using the network's structure: a network is a composition of simple steps, the derivative of each step is easy, and the chain rule assembles them.

So we give each step a name on both sides of it.

Exercise 2.1 (Give the intermediate quantities names). The network chapter wrote the whole model in two lines. To differentiate it, we break the two lines into four, so that every place a function is applied has a name on each side of it:

$$\tilde{a} = W^\top x, \quad \tilde{h} = f(\tilde{a}), \quad b = U^\top \begin{bmatrix} 1 \\ \tilde{h} \end{bmatrix}, \quad y = g(b).$$

$\tilde{a}$ and b are the **pre-activations**, $\tilde{h}$ and y the **post-activations**.

Give the shape of each of the four vectors, for p inputs^a, L hidden units and C outputs.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Solution

Solution 2.1. $x \in \mathbb{R}^{p+1}$ (the $+1$ is the constant^a input^b), $\tilde{a} \in \mathbb{R}^L$, $\tilde{h} \in \mathbb{R}^L$, $b \in \mathbb{R}^C$, $y \in \mathbb{R}^C$. W is $(p+1) \times L$ and U is $(L+1) \times C$; the bracket $[1; \tilde{h}]$ supplies U 's extra row.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

^b<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Exercise 2.2 (Give the intermediate quantities names (part 2)). Use the setup of Exercise 2.1.

The equation $\tilde{h} = f(W^\top \tilde{a})$ says the same thing as the first two equations together. Give a reason to keep $\tilde{a}$ as a separate name anyway, in terms of what the chain rule will need.

Solution

Solution 2.2. Because we compute the derivative with respect to the pre-activation once and use it twice.

Writing $\delta_j^a = \partial \mathcal{L} / \partial a_j$, every weight in column j of W has

$$\frac{\partial \mathcal{L}}{\partial W_{ij}} = \delta_j^a x_i,$$

so the L numbers in $\tilde{\delta}^a$ give all $(p+1)L$ derivatives in W . We also pass the same $\tilde{\delta}^a$ further back in a deeper network. If you never name $\tilde{a}$, you have no name for the quantity the algorithm^a is actually propagating — which is why the method is named after it.

^a<https://morrison-lab.github.io/mds/expressions-algorithms.html#def-algorithm>

Exercise 2.3 (Give the intermediate quantities names (part 3)). Use the setup of Exercise 2.1.

The backward pass will reuse quantities from the forward pass rather than recompute them. Before

deriving it, **predict** which of $\tilde{x}, \tilde{a}, \tilde{h}, \tilde{b}, \tilde{y}$ have to survive until the backward pass and which you can discard as soon as you compute the next line, and say how much memory your answer costs for a minibatch of B examples. Exercise 3.1 and Exercise 4.1 will settle it, and predicting first gives you something to check them against.

Solution

Solution 2.3. Deriving Exercise 3.1 and Exercise 4.1 settles it. The backward pass needs:

- $\tilde{x}$, to form $\partial\mathcal{L}/\partial W$;
- $\tilde{h}$, to form $\partial\mathcal{L}/\partial U$;
- $\tilde{y}$, to start the recursion;
- and $f'(\tilde{a})$, which for the logistic sigmoid^a is recoverable from $\tilde{h}$ alone, since $f'(a) = h(1 - h)$.

We can discard $\tilde{b}$ the moment we compute $\tilde{y}$, and $\tilde{a}$ too whenever f' is expressible in $\tilde{h}$ — which is why implementations usually store the post-activations and not the pre-activations.

For a minibatch of B examples that is $B(p + L + C + 1)$ numbers, against $P = (p + 1)L + (L + 1)C$ for the parameters themselves. With $p = 784$, $L = 100$, $C = 10$ and $B = 128$, the stored activations come to 114,560 numbers and the parameters to 79,510 — the activations are the larger of the two, by a factor of 1.44. Holding both at once therefore costs 2.44 times what the model alone does, and in a deep network it is worse, because we hold every layer's activations until the backward pass reaches them.

^a<https://morrison-lab.github.io/mds/functions.html#def-logistic-function>

Exercise 2.4 (Give the intermediate quantities names (part 4)). Use the setup of Exercise 2.1.

Separate the quantities that depend on the training example from those that do not, and say what that separation implies about which of them a minibatch has B copies of.

Solution

Solution 2.4. W and U do not depend on the example; $\tilde{x}, \tilde{a}, \tilde{h}, \tilde{b}, \tilde{y}$ all do. A minibatch therefore carries B copies of each activation and **one** copy of each parameter.

That asymmetry is the one Exercise 12.7 turns into a number. Whether reading each weight once for B examples gives any real benefit depends on figures that exercise supplies, so note the asymmetry here and leave the conclusion until then.

Exercise 2.5 (Give the intermediate quantities names (part 5)). Use the setup of Exercise 2.1.

Suppose f were the identity. Which of the four names would still be worth keeping, and what does your answer say about *why* the nonlinearity is what makes the bookkeeping necessary?

Solution

Solution 2.5. If f were the identity then $\tilde{h} = \tilde{a}$ and one of the two names is redundant — and more than that, the whole two-layer structure collapses, since $\tilde{b} = U^\top[1; W^\top \tilde{x}]$ is affine in $\tilde{x}$ and could be replaced by a single matrix. There would be nothing to propagate through.

The bookkeeping exists because f lies between two linear maps and does not commute with them. Every complication in this chapter — the elementwise factor $f'(\tilde{a})$, the stored activations, the vanishing gradients of Exercise 8.1 — traces^a back to that one function, and so does every gain in representational power from the neural networks chapter.

^a<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-trace>

3 The last layer

Differentiation runs in the opposite direction to the computation: the loss is at the end, so the derivative that is easiest to write is the one nearest to it.

We carry over one result rather than rederive it. The neural networks chapter showed that a softmax output with a cross-entropy loss — and equally a sigmoid output with a binary cross-entropy loss, and a linear output with the **halved** squared error $\frac{1}{2}\|y - \tilde{t}\|^2$ — all give the same derivative at the output pre-activation:

$$\begin{aligned}\delta_{\tilde{}}^b &= \frac{\partial \mathcal{L}}{\partial b} \\ &= y - \tilde{t}\end{aligned}$$

Prediction minus target. Each of the three matched pairs produces it, which is why those pairs are the standard ones. It is the only fact about the loss that the rest of this chapter uses. The $\frac{1}{2}$ cancels the 2 that differentiating a square produces, which is why the squared-error convention includes it. Exercise 3.7 drops it and shows the 2 reappear.

Exercise 3.1 (Differentiate the last layer first). The neural networks chapter established the one derivative this chapter takes as given: with a softmax output and the cross-entropy loss, the derivative of the loss with respect to the output **pre-activation** is

$$\begin{aligned}\delta_{\tilde{}}^b &= \frac{\partial \mathcal{L}}{\partial b} \\ &= y - \tilde{t},\end{aligned}$$

where $\tilde{t}$ is the one-hot target. Prediction minus target, with the softmax and the logarithm cancelling each other out. Everything else in this chapter is the chain rule applied to that.

Write $b_c = \sum_j U_{jc} [1; \tilde{h}]_j$ and differentiate it with respect to a single entry $U_{jc'}$. Be explicit about when the answer is zero.

Solution

Solution 3.1. b_c involves only column c of U , so

$$\frac{\partial b_c}{\partial U_{jc'}} = \begin{cases} [1; \tilde{h}]_j & c' = c \\ 0 & c' \neq c. \end{cases}$$

The zero is the useful half: changing a weight in column 3 cannot affect output 5 at all.

Exercise 3.2 (Differentiate the last layer first (part 2)). Use the setup of Exercise 3.1. Combine Exercise 3.1 with $\delta_{\tilde{}}^b$ to get $\partial \mathcal{L} / \partial U_{jc}$.

Solution

Solution 3.2. The chain rule sums over the outputs, and Solution 3.1 makes every term zero except one:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial U_{jc}} &= \sum_{c'} \delta_{\tilde{}}^b \frac{\partial b_{c'}}{\partial U_{jc}} \\ &= \delta_c^b [1; \tilde{h}]_j \\ &= (y_c - \tilde{t}_c) [1; \tilde{h}]_j.\end{aligned}$$

Read it as a product of two things the forward pass already has: **how wrong output c was**, and **how active unit j was**. The gradient moves a weight in proportion to both. That is the form of the learning rule Donald Hebb proposed in 1949 — strengthen a connection between two units that are

active together — with the second factor replaced by an error rather than an activity, so the connection that changes is the one between an active input^a and a *wrong* output. The resemblance is structural, and we should not take it further. Hebb’s rule is local and unsupervised, and this one needs a loss.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Exercise 3.3 (Differentiate the last layer first (part 3)). Use the setup of Exercise 3.1. Write the whole matrix $\partial\mathcal{L}/\partial U$ in one expression^a, with no indices, and check that its shape matches U .

^a<https://morrison-lab.github.io/mds/expressions-algorithms.html#def-expression>

Solution

Solution 3.3. That is an outer product^a:

$$\frac{\partial\mathcal{L}}{\partial U} = \begin{bmatrix} 1 \\ \tilde{h} \end{bmatrix} (\tilde{\delta}^b)^\top.$$

The shapes work out: $(L+1) \times 1$ times $1 \times C$ gives $(L+1) \times C$, which is U ’s shape.

^a<https://morrison-lab.github.io/mds/linear-algebra-inner-products.html#def-outer-product>

Exercise 3.4 (Differentiate the last layer first (part 4)). Use the setup of Exercise 3.1. What is the rank^a of that matrix, for a single training example?

^a<https://morrison-lab.github.io/mds/linear-algebra-subspaces.html#def-rank>

Solution

Solution 3.4. An outer product^a of two non-zero vectors has rank^b 1.

^a<https://morrison-lab.github.io/mds/linear-algebra-inner-products.html#def-outer-product>

^b<https://morrison-lab.github.io/mds/linear-algebra-subspaces.html#def-rank>

Exercise 3.5 (Differentiate the last layer first (part 5)). Use the setup of Exercise 3.1. What does that rank^a say about how much a single example can change U ?

^a<https://morrison-lab.github.io/mds/linear-algebra-subspaces.html#def-rank>

Solution

Solution 3.5. So a single example can only push U along one direction in a $(L+1)C$ -dimensional space, no matter how large the step. Fitting U properly needs at least $\min(L+1, C)$ independent^a examples before the accumulated gradient can even have full rank^b. This is one concrete reason the stochastic steps of gradient descent have to be numerous and small: each one is a rank-one correction, and the useful direction only emerges from their average.

^a<https://morrison-lab.github.io/pds/independence.html#def-indpt>

^b<https://morrison-lab.github.io/mds/linear-algebra-subspaces.html#def-rank>

Exercise 3.6 (Differentiate the last layer first (part 6)). Use the setup of Exercise 3.1. Row 0 of U multiplies the constant^a 1. Write its gradient, and say in one sentence what quantity it is the gradient of.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

Solution

Solution 3.6. Setting $j = 0$, where $[1; \tilde{h}]_0 = 1$:

$$\frac{\partial \mathcal{L}}{\partial U_{0c}} = y_c - t_c.$$

That row is the output **bias**^a, and its gradient is the prediction error^b itself, unweighted by any activation. It is the one parameter that moves on every example regardless of what the hidden layer did.

^a<https://morrison-lab.github.io/sds/estimation.html#def-bias>

^b<https://morrison-lab.github.io/sds/prediction.html#def-prediction-error>

Exercise 3.7 (Differentiate the last layer first (part 7)). Use the setup of Exercise 3.1.

Suppose the task were regression instead: g the identity and $\mathcal{L} = \|y - \tilde{t}\|^2$. Recompute $\tilde{\delta}^b$ and say which parts of your answers to Exercise 3.2 and Exercise 3.3 change.

Solution

Solution 3.7. With g the identity, $y = b$, and

$$\begin{aligned}\tilde{\delta}^b &= \frac{\partial}{\partial b} \|b - \tilde{t}\|^2 \\ &= 2(y - \tilde{t}).\end{aligned}$$

Parts 2 and 3 do not change at all beyond the factor of 2 — the outer product^a is the same expression^b with a different $\tilde{\delta}^b$ in it.

This is the first sign of the structure the whole chapter has: **the layer does not know what loss it is serving**. The single vector $\tilde{\delta}^b$ summarizes everything above the pre-activation b , and the layer's own derivative is the same formula whatever produced it. That makes the recursion in Exercise 6.1 possible, and lets a library offer losses and layers as separate, interchangeable pieces.

^a<https://morrison-lab.github.io/mds/linear-algebra-inner-products.html#def-outer-product>

^b<https://morrison-lab.github.io/mds/expressions-algorithms.html#def-expression>

i Watch

- Sanderson applies the chain rule to a network, from 0:38^a [1, Ch. 4].
- Starmer explains the chain rule^b [5], optimizes the last layer^c [6] of a small network, and differentiates cross-entropy^d [7] and the softmax^e [8].

^a<https://www.youtube.com/watch?v=tIeHLnjs5U8&t=38s>

^b<https://www.youtube.com/watch?v=w1myxrtQHQ>

^c<https://www.youtube.com/watch?v=iyn2zdALii8>

^d<https://www.youtube.com/watch?v=xBEh66V9gZo>

^e<https://www.youtube.com/watch?v=M59JEIEPgIg>

4 One layer back

U was the easy matrix: it sits directly beneath the loss. W is underneath U and underneath f , and it holds almost all of the parameters.

Passing the derivative back through those two obstacles is the whole of back-propagation [9]. The first is a matrix, and passing back through it costs a sum. The second is an elementwise function, and passing back through it costs a multiplication.

Exercise 4.1 (Push the error back through the hidden layer). Exercise 3.1 stopped at U . The other matrix, W , is the one holding 99% of the parameters, and W is separated from the loss by two things: the multiplication by U , and the activation function f . This exercise handles the multiplication and the activation function, one at a time.

Throughout, $\delta^b = y - t$ is known from Exercise 3.1.

Unlike a weight in U , the hidden value h_j affects **every** output. Write $\partial\mathcal{L}/\partial h_j$ as a sum over c , and simplify that sum using $b_c = \sum_{j'} U_{j'c}[1; \tilde{h}]_{j'}$.

Solution

Solution 4.1. h_j reaches the loss through all C outputs, so the chain rule sums over them:

$$\begin{aligned}\frac{\partial\mathcal{L}}{\partial h_j} &= \sum_{c=1}^C \delta_c^b \frac{\partial b_c}{\partial h_j} \\ &= \sum_{c=1}^C \delta_c^b U_{jc}.\end{aligned}$$

Exercise 4.2 (Push the error back through the hidden layer (part 2)). Use the setup of Exercise 4.1. Write your answer to Exercise 4.1 as a matrix-vector product, $\tilde{\delta}^h = ?$. The matrix is not quite U : say which part of U it leaves out and why.

Solution

Solution 4.2. That sum is row j of U against $\tilde{\delta}^b$, so stacking over j ,

$$\tilde{\delta}^h = U_{1:L,:} \tilde{\delta}^b,$$

where $U_{1:L,:}$ is U with its row 0 deleted. That row multiplies the constant^a 1, not any h_j , so it contributes to no h_j 's derivative. Keeping it would produce an $(L+1)$ -vector whose first entry is the derivative of the loss with respect to a constant, which is not a quantity that exists.

(The constant sits first throughout this chapter, in position 0, as the neural networks chapter sets it up and as linear regression pads its inputs^b, so row j of U multiplies h_j and row 0 is the bias^c.)

This is the step that gives the method its name: the same matrix U that carried activations forward carries errors backward, transposed.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

^b<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^c<https://morrison-lab.github.io/sds/estimation.html#def-bias>

Exercise 4.3 (Push the error back through the hidden layer (part 3)). Use the setup of Exercise 4.1. Now pass back through f , using $h_j = f(a_j)$. Give δ_j^a , and explain why this step produces no sum, where Exercise 4.1 did. Express the result in vector form with the elementwise product $\odot$.

Solution

Solution 4.3. f is applied elementwise, so a_j affects h_j and nothing else:

$$\delta_j^a = \delta_j^h f'(a_j), \quad \tilde{\delta}^a = \tilde{\delta}^h \odot f'(\tilde{a}).$$

There is no sum because the Jacobian $\partial h_j / \partial a_j$ is **diagonal**^a. Solution 4.1 needed a sum because $\partial b / \partial \tilde{h} = (U_{1:L,:})^\top$ is a full $C \times L$ matrix. Elementwise functions are cheap to differentiate for exactly this reason, which is why we apply activation functions coordinate by coordinate rather than as a general $\mathbb{R}^L \rightarrow \mathbb{R}^L$ map.

^a<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-diagonal-matrix>

Exercise 4.4 (Push the error back through the hidden layer (part 4)). Use the setup of Exercise 4.1. Finally, $\tilde{a} = W^\top x$ has exactly the shape Exercise 3.1 already handled. Give $\partial\mathcal{L}/\partial W_{ij}$ and then the whole matrix without indices.

Solution

Solution 4.4. $a_j = \sum_i W_{ij}x_i$, so $\partial a_j/\partial W_{ij} = x_i$ and

$$\frac{\partial\mathcal{L}}{\partial W_{ij}} = \delta_j^a x_i, \quad \frac{\partial\mathcal{L}}{\partial W} = \tilde{x}(\tilde{\delta}^a)^\top.$$

The same outer-product^a shape as U 's gradient, with x in place of $[1; h]$ and δ^a in place of $\tilde{\delta}^b$. The input^b to a layer and the error at its output are all a layer's gradient ever needs.

^a<https://morrison-lab.github.io/mds/linear-algebra-inner-products.html#def-outer-product>

^b<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Exercise 4.5 (Push the error back through the hidden layer (part 5)). Use the setup of Exercise 4.1. Lay the four results out in order and check that the shapes compose: $(C) \rightarrow (L) \rightarrow (L) \rightarrow (p+1) \times L$.

Solution

Solution 4.5. In order:

$$\tilde{\delta}^b = y - t \in \mathbb{R}^C \rightarrow \tilde{\delta}^h = U_{1:L,:} \tilde{\delta}^b \in \mathbb{R}^L \rightarrow \tilde{\delta}^a = \tilde{\delta}^h \odot f'(\tilde{a}) \in \mathbb{R}^L \rightarrow \tilde{x}(\tilde{\delta}^a)^\top \in \mathbb{R}^{(p+1) \times L}.$$

$U_{1:L,:}$ is $L \times C$ against a C -vector, giving an L -vector; the elementwise product preserves that; and $(p+1) \times 1$ against $1 \times L$ gives W 's shape.

Exercise 4.6 (Push the error back through the hidden layer (part 6)). Use the setup of Exercise 4.1. For $f = \text{expit}$, write $f'(a_j)$ in terms of h_j alone, give its largest possible value, and say what happens to δ_j^a when unit j is strongly on or strongly off.

Solution

Solution 4.6. $\text{expit}'(a) = \text{expit}(a)(1 - \text{expit}(a)) = h_j(1 - h_j)$, so the derivative is recoverable from the stored activation and a need not be kept.

It is largest at $h_j = 1/2$, where it equals $1/4$, and it goes to zero at both ends. A unit that is strongly on ($h_j \approx 1$) or strongly off ($h_j \approx 0$) has $\delta_j^a \approx 0$, so **none of the weights feeding it move**, however wrong the network's output was.

That is the saturation problem. The same $\text{expit}' \leq 1/4$ made the logic gates of the neural networks chapter work so cleanly — a unit driven to ± 10 gives a clear answer precisely because it has stopped responding to its input^a — and Exercise 8.1 shows what it costs once such units are stacked.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

i Watch

- Hutchinson covers backprop for a single hidden layer^a (28 min) [3].
- Starmer carries the chain rule back a layer^b [10].

^ahttps://facultyweb.cs.wvu.edu/~hutchib2/video_lectures/data371/#backprop

^b<https://www.youtube.com/watch?v=GKZoOHXGcLo>

5 The algorithm

Collecting the results, for one training example and one hidden layer:

i Backpropagation algorithm (one hidden layer)

Forward. $\tilde{a} = W^\top x$, $\tilde{h} = f(\tilde{a})$, $b = U^\top [1; \tilde{h}]$, $y = g(b)$.

Backward.

$$\delta^b = y - t$$

$$\frac{\partial \mathcal{L}}{\partial U} = [1; \tilde{h}] (\delta^b)^\top$$

$$\tilde{\delta}^a = (U_{1:L,:} \tilde{\delta}^b) \odot f'(\tilde{a})$$

$$\frac{\partial \mathcal{L}}{\partial W} = x (\tilde{\delta}^a)^\top$$

Each of the four lines is a matrix-vector product, an outer product, or an elementwise multiplication. They share a shape: **every parameter gradient is the layer's input times the error at the layer's output**. That is the pattern the next section turns into a loop.

Exercise 5.1 (Count what the backward pass actually costs). Exercise 1.1 measured the cost of the obvious method and set a target: a gradient in a small constant number of forward passes. Exercise 3.1 and Exercise 4.1 produced an algorithm. Now check the algorithm against the target.

Use the same network: $p = 784$, $L = 100$, $C = 10$, so W is 785×100 , U is 101×10 , and one forward pass is 79,510 multiply-accumulates.

Count the multiply-accumulates in each of the four backward steps separately: $\delta^h = U_{1:L,:} \delta^b$, the outer product^a for $\partial \mathcal{L} / \partial U$, the elementwise product giving $\tilde{\delta}^a$, and the outer product for $\partial \mathcal{L} / \partial W$.

^a<https://morrison-lab.github.io/mds/linear-algebra-inner-products.html#def-outer-product>

Solution

Solution 5.1. Each step is a product whose cost is the number of entries it touches:

step	shape	multiply-accumulates
$\tilde{\delta}^h = U_{1:L,:} \tilde{\delta}^b$	$(100 \times 10)(10)$	1,000
$\partial \mathcal{L} / \partial U = [1; \tilde{h}] (\tilde{\delta}^b)^\top$	$(101)(10)$	1,010
$\tilde{\delta}^a = \tilde{\delta}^h \odot f'(\tilde{a})$	(100)	100
$\partial \mathcal{L} / \partial W = x (\tilde{\delta}^a)^\top$	$(785)(100)$	78,500

Exercise 5.2 (Count what the backward pass actually costs (part 2)). Use the setup of Exercise 5.1. Total them, and give the ratio of backward cost to forward cost. Which single step dominates, and why is that the same step that dominated the forward pass?

Solution

Solution 5.2. The total is 80,610, against 79,510 forward, a ratio of 1.014. The backward pass costs essentially the same as the forward pass.

$\partial\mathcal{L}/\partial W$ dominates both, at 78,500 of each total, and for the same reason in both directions: it has one entry per parameter in W , and W holds 98.7% of the parameters. Forward or backward, the work is proportional to the number of weights, because each pass touches each weight once.

Exercise 5.3 (Count what the backward pass actually costs (part 3)). Use the setup of Exercise 5.1. Give the cost of forward-plus-backward in units of forward passes, and compare it with the $2P$ forward passes of Exercise 1.1. By what factor has the gradient got cheaper?

Solution

Solution 5.3. Forward plus backward is 160,120 multiply-accumulates, or 2.01 forward passes. Common estimates^a usually say “about three”, and that figure is a count of matrix products^b rather than an allowance for overheads: one product forward, two back, as Solution 5.5 works out. This network comes to two rather than three because layer 1 skips one of its two backward products. $\tilde{\delta}^x$ would be the error on the input^c, and the input is data, not a parameter — we fit nothing below it, so there is nothing to propagate to. Layer 1 holds 98.7% of the weights, so skipping its input-error product removes almost exactly one forward pass’ worth of work: restoring it gives 238,620 multiply-accumulates, or 3.0011 forward passes. A deep network needs the full three, because only its first layer skips that product. Exercise 1.1 needed $2P = 159,020$ forward passes, or 1.26×10^{10} multiply-accumulates. The gradient is cheaper by a factor of

$$\frac{1.26 \times 10^{10}}{160,120} \approx 79,000.$$

^a<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

^b<https://morrison-lab.github.io/mds/linear-algebra-matrices.html#def-matrix-mult>

^c<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Exercise 5.4 (Count what the backward pass actually costs (part 4)). Use the setup of Exercise 5.1. The quadratic in Exercise 1.1 has become linear. Say where it went: what does back-propagation compute **once** that the numerical method recomputed for every parameter?

Solution

Solution 5.4. It computes $\tilde{\delta}^b$ and $\tilde{\delta}^a$ once, and every parameter’s derivative is a product of one of those numbers with one stored activation.

The numerical method does not share any work. To learn about weight W_{ij} it runs the entire network twice, and the 79,509 other weights take part in that computation without teaching us anything about them. Back-propagation does the shared part once — the L numbers in $\tilde{\delta}^a$ — and then spends exactly one multiplication per parameter.

That is the whole idea, stated as a rule: **the cost is quadratic when the work per parameter is a function evaluation, and linear when the work per parameter is a multiplication.**

Exercise 5.5 (Count what the backward pass actually costs (part 5)). Use the setup of Exercise 5.1. None of these counts depended on f , g , or the loss. State the general rule for the cost of a backward pass through any layer whose forward pass is a matrix product^a, and say what this implies for a network of K layers.

^a<https://morrison-lab.github.io/mds/linear-algebra-matrices.html#def-matrix-mult>

Solution

Solution 5.5. A layer computing $\tilde{v} = M^\top \tilde{u}$ costs $|M|$ multiply-accumulates forward. Backward it costs $|M|$ for the input^a error $\tilde{\delta}^u = M \tilde{\delta}^v$ and another $|M|$ for the parameter gradient $u(\tilde{\delta}^v)^\top$ — so a backward pass is about **twice** a forward pass per layer, and the whole of training is a small constant^b times the cost of prediction.

For K layers, both passes are a sum over layers of $|M^{(k)}|$, so everything stays linear in the total parameter count however deep the network gets. Depth costs memory, as Exercise 2.3 showed, and it costs numerical trouble, as Exercise 8.1 shows. It does not cost arithmetic.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^b<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

i Watch

The CS231N lecture on backpropagation builds the algorithm on a computational graph, from 35:52^a and then the backpropagation algorithm, from 40:36^b [11], Lecture 4.

^a<https://www.youtube.com/watch?v=25zD5qJHYsk&t=2152s>

^b<https://www.youtube.com/watch?v=25zD5qJHYsk&t=2436s>

i Watch

The CS229 lecture covers backpropagation and improving neural networks^a [12], Lecture 11.

^a<https://www.youtube.com/watch?v=zUazLXZZA2U>

6 Depth

Nothing in the derivation used the fact that there was one hidden layer. The backward step took a derivative with respect to one pre-activation and produced a derivative with respect to the pre-activation below it — the same kind of object, one layer down.

A procedure that turns an object into another object of the same kind is a recursion, and running it K times makes a network *deep*.

Exercise 6.1 (Do it again, K times). Nothing in Exercise 4.1 used the fact that there was only one hidden layer. The derivation passed back through a matrix, then through an activation function, and arrived at a quantity of exactly the same kind it started with. That is a recursion, and we can now write it down.

Take K hidden layers, with widths $L_0 = p$, $L_1, \dots, L_K$, and $L_{K+1} = C$. Write $\tilde{h}^{(0)} = \tilde{x}$ and, for $k = 1, \dots, K + 1$,

$$\tilde{a}^{(k)} = (W^{(k)})^\top \begin{bmatrix} 1 \\ \tilde{h}^{(k-1)} \end{bmatrix}, \quad \tilde{h}^{(k)} = f(\tilde{a}^{(k)}),$$

with g in place of f at the last layer, so $y = \tilde{h}^{(K+1)}$.

Give the shape of $W^{(k)}$ and the total parameter count.

Solution

Solution 6.1. $W^{(k)}$ maps a layer of width L_{k-1} plus a constant^a to a layer of width L_k , so it is $(L_{k-1} + 1) \times L_k$, and

$$P = \sum_{k=1}^{K+1} (L_{k-1} + 1) L_k.$$

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

Exercise 6.2 (Do it again, K times (part 2)). Use the setup of Exercise 6.1.

Write the recursion that takes $\tilde{\delta}^{(k+1)} = \partial \mathcal{L} / \partial a^{(k+1)}$ to $\tilde{\delta}^{(k)}$, and say which parts of Exercise 4.1 each piece comes from.

Solution

Solution 6.2. Both steps of Exercise 4.1, in the same order:

$$\tilde{\delta}^{(k)} = \underbrace{\left(W_{1:L_k, :}^{(k+1)} \tilde{\delta}^{(k+1)} \right)}_{\text{across the matrix, @sol-deep-recursion-part2}} \odot \underbrace{f'(a^{(k)})}_{\text{across the activation, @sol-deep-recursion-part3}}.$$

The row deletion is the same one: $W^{(k+1)}$'s row 0 multiplies the constant^a 1, not any unit of layer k . The recursion starts at the top with $\tilde{\delta}^{(K+1)} = y - \tilde{t}$ from Exercise 3.1. That first value is special only because the loss enters there.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-constant-function>

Exercise 6.3 (Do it again, K times (part 3)). Use the setup of Exercise 6.1.

Give $\partial \mathcal{L} / \partial W^{(k)}$ in terms of $\tilde{\delta}^{(k)}$ and the stored activations.

Solution

Solution 6.3. Exactly the outer product^a of Exercise 4.4, with the layer's own input^b in front:

$$\frac{\partial \mathcal{L}}{\partial W^{(k)}} = \begin{bmatrix} 1 \\ h^{(k-1)} \end{bmatrix} \left(\tilde{\delta}^{(k)} \right)^\top.$$

Every layer's gradient has the same form: **the input to the layer, times the error at its output.**

^a<https://morrison-lab.github.io/mds/linear-algebra-inner-products.html#def-outer-product>

^b<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Exercise 6.4 (Do it again, K times (part 4)). Use the setup of Exercise 6.1.

Write the whole algorithm^a as two loops, forward then backward, and say what has to be kept between them.

^a<https://morrison-lab.github.io/mds/expressions-algorithms.html#def-algorithm>

Solution

Solution 6.4. Forward, storing as you go:

for $k = 1 \dots K + 1$: compute $a^{(k)}$, then $\tilde{h}^{(k)}$, and keep $\tilde{h}^{(k-1)}$.

Backward, starting from $\tilde{\delta}^{(K+1)} = y - \tilde{t}$:

for $k = K + 1 \dots 1$: record $\partial \mathcal{L} / \partial W^{(k)} = [1; \tilde{h}^{(k-1)}] (\tilde{\delta}^{(k)})^\top$, then, if $k > 1$, form $\tilde{\delta}^{(k-1)} = \left(W_{1:L_{k-1}, :}^{(k)} \tilde{\delta}^{(k)} \right) \odot f'(a^{(k-1)})$.

Between the two loops we keep every layer's post-activation $\tilde{h}^{(k)}$ — which is also enough to recover $f'(a^{(k)})$ when f is the logistic sigmoid^a, since $\text{expit}'(a) = h(1 - h)$.

The order inside the backward loop matters. The parameter gradient uses $\delta^{(k)}$, and only then do we use $W^{(k)}$ to produce $\delta^{(k-1)}$. Using the old $W^{(k)}$ for that second step is essential. An implementation that updated $W^{(k)}$ before propagating through it would be differentiating a network that never existed.

^a<https://morrison-lab.github.io/mds/functions.html#def-logistic-function>

Exercise 6.5 (Do it again, K times (part 5)). Use the setup of Exercise 6.1.

There is an alternative that never runs backwards: start at the inputs^a and propagate derivatives with respect to one chosen parameter forward through the network, the way the values themselves travel. Work out what that costs for all P parameters, and use the answer to explain why the algorithm^b is called *back-propagation*.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^b<https://morrison-lab.github.io/mds/expressions-algorithms.html#def-algorithm>

Solution

Solution 6.5. Propagating forward, you would carry $\partial \tilde{h}^{(k)} / \partial \theta$ for one chosen parameter θ alongside the values, at roughly the cost of the forward pass itself — and you would have to do it again for each of the P parameters. That is $O(P)$ forward passes — the same order as Exercise 1.1, which needed $2P$ of them, and it is no simpler to write.

The asymmetry is the reason for the direction. A forward sweep computes the derivatives of **many outputs with respect to one input**^a; a backward sweep computes the derivatives of **one output with respect to many inputs**^b. Training has one output — the scalar loss — and P inputs — the parameters. That is the case where going backwards wins, and it wins by a factor of P .

Both sweeps are standard and both are used: automatic-differentiation libraries call them forward mode and reverse mode, and choose between them by comparing the number of inputs with the number of outputs. Back-propagation is reverse mode, specialized to the one function form that machine learning cares about.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^b<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

7 What depth costs

The arithmetic stays linear in the parameter count however deep the network is, which makes depth look free. It is not.

The recursion multiplies δ by a derivative of the activation function at every layer it passes, and the logistic sigmoid's derivative is never larger than $1/4$. Over ten layers that is a factor of 10^{-6} . This is a large part of why networks deeper than two or three layers resisted training from the 1980s until the late 2000s, when better initialization, unsupervised pre-training and then the rectified linear unit (ReLU) each addressed one part of it.

8 Measuring it

The bound of Section 7 is for the worst case. Whether it matters in practice is a question about real initialized networks, so we measure it on 200 sigmoid networks with 10 hidden layers of 20 units, 20 inputs and 3 classes, each weight drawn from $\mathcal{N}(0, 1/(\text{fan-in} + 1))$, each fed one standard normal input against one uniformly chosen one-hot target under softmax and cross-entropy. We report the Frobenius norm of $\partial \mathcal{L} / \partial W^{(k)}$, averaged over the 200 networks, together with the geometric mean of the ratios between consecutive layers.

Each version draws its own random numbers, so the three agree to about two significant figures rather than exactly. Replacing the sigmoid with $\max(a, 0)$ gives the ReLU figures of Exercise 8.7.

8.0.1 Python

```
import numpy as np

K, L, D, C, N = 10, 20, 20, 3, 200
rng = np.random.default_rng(0)
widths = [D] + [L] * K + [C]
norms = np.zeros(K + 1)

for _ in range(N):
    W = [rng.normal(0, 1 / np.sqrt(widths[k] + 1), (widths[k] + 1, widths[k + 1]))
          for k in range(K + 1)]
    post = [np.concatenate(([1.0], rng.normal(0, 1, D)))]
    pre = []
    for k in range(K + 1):
        a = W[k].T @ post[k]
        pre.append(a)
        if k < K:
            post.append(np.concatenate(([1.0], 1 / (1 + np.exp(-a)))))

    e = np.exp(pre[K] - pre[K].max())
    t = np.zeros(C)
    t[rng.integers(C)] = 1.0
    d = e / e.sum() - t

    for k in range(K, -1, -1):
        norms[k] += np.linalg.norm(np.outer(post[k], d))
        if k > 0:
            h = post[k][1:]
            d = (W[k][1:, :] @ d) * h * (1 - h)

norms /= N
r = norms[1:] / norms[:-1]
print(f"layer 1 {norms[0]:.2e}")
```

layer 1 1.11e-06

```
print(f"output {norms[K]:.2f}")
```

output 2.12

```
print(f"ratio {norms[0] / norms[K]:.1e}")
```

ratio 5.2e-07

```
print(f"geomean {np.exp(np.mean(np.log(r))):.2f}")
```

geomean 4.25

8.0.2 R

```
# nolint start: object_name_linter. K, L, D, C, N and W match the prose.
K <- 10
L <- 20
D <- 20
C <- 3
N <- 200
widths <- c(D, rep(L, K), C)
set.seed(0)
```

```

norms <- numeric(K + 1)

for (n in seq_len(N)) {
  W <- lapply(seq_len(K + 1), function(k) {
    matrix(rnorm((widths[k] + 1) * widths[k + 1]), 0, 1 / sqrt(widths[k] + 1)),
          widths[k] + 1, widths[k + 1])
  })
  post <- list(c(1, rnorm(D)))
  pre <- list()
  for (k in seq_len(K + 1)) {
    a <- as.vector(t(W[[k]]) %*% post[[k]])
    pre[[k]] <- a
    if (k <= K) post[[k + 1]] <- c(1, 1 / (1 + exp(-a)))
  }

  e <- exp(pre[[K + 1]] - max(pre[[K + 1]]))
  t_vec <- numeric(C)
  t_vec[sample.int(C, 1)] <- 1
  d <- e / sum(e) - t_vec

  for (k in seq(K + 1, 1)) {
    norms[k] <- norms[k] + sqrt(sum(outer(post[[k]], d)^2))
    if (k > 1) {
      h <- post[[k]][-1]
      d <- as.vector(W[[k]][-1, , drop = FALSE] %*% d) * h * (1 - h)
    }
  }
}

norms <- norms / N
r <- norms[-1] / head(norms, -1)
cat(sprintf("layer 1 %.2e\n", norms[1]))

layer 1 1.13e-06

cat(sprintf("output %.2f\n", norms[K + 1]))

output 2.06

cat(sprintf("ratio %.1e\n", norms[1] / norms[K + 1]))

ratio 5.5e-07

cat(sprintf("geomean %.2f\n", exp(mean(log(r)))))

geomean 4.23
# nolint end

```

8.0.3 Julia

```

using Random, LinearAlgebra, Statistics, Printf

K, L, D, C, N = 10, 20, 20, 3, 200;
widths = [D; fill(L, K); C];
rng = MersenneTwister(0);
norms = zeros(K + 1);

for _ in 1:N

```

```

W = [randn(rng, widths[k] + 1, widths[k+1]) / sqrt(widths[k] + 1) for k in 1:K+1]
post = [[1.0; randn(rng, D)]]
pre = Vector{Vector{Float64}}{ }
for k in 1:K+1
    a = W[k]' * post[k]
    push!(pre, a)
    k <= K && push!(post, [1.0; 1 ./ (1 .+ exp.(-a))])
end

e = exp.(pre[K+1] .- maximum(pre[K+1]))
t = zeros(C)
t[rand(rng, 1:C)] = 1.0
d = e ./ sum(e) .- t

for k in K+1:-1:1
    norms[k] += norm(post[k] * d')
    if k > 1
        h = post[k][2:end]
        d = (W[k][2:end, :] * d) .* h .* (1 .- h)
    end
end
end

norms ./= N;
r = norms[2:end] ./ norms[1:end-1];
@printf("layer 1 %.2e\n", norms[1])

```

layer 1 1.06e-06

```
@printf("output  %.2f\n", norms[K+1])
```

output 2.06

```
@printf("ratio  %.1e\n", norms[1] / norms[K+1])
```

ratio 5.2e-07

```
@printf("geomean %.2f\n", exp(mean(log.(r))))
```

geomean 4.25

The Python measurement above as a function of the activation and its derivative, so the exercises below can rerun it with another activation.
Run with the sigmoid it reproduces the Python tab's norms (checked below).

```

def measure(act, dact, K=10, L=20, D=20, C=3, N=200, seed=0):
    rng = np.random.default_rng(seed)
    widths = [D] + [L] * K + [C]
    norms = np.zeros(K + 1)
    for _ in range(N):
        W = [rng.normal(0, 1 / np.sqrt(widths[k] + 1),
                        (widths[k] + 1, widths[k + 1]))
              for k in range(K + 1)]
        post = [np.concatenate(([1.0], rng.normal(0, 1, D)))]
        pre = []
        for k in range(K + 1):
            a = W[k].T @ post[k]
            pre.append(a)
            if k < K:
                post.append(np.concatenate(([1.0], act(a))))

```

```

    e = np.exp(pre[K] - pre[K].max())
    t = np.zeros(C)
    t[rng.integers(C)] = 1.0
    d = e / e.sum() - t
    for k in range(K, -1, -1):
        norms[k] += np.linalg.norm(np.outer(post[k], d))
        if k > 0:
            h = post[k][1:]
            d = (W[k][1:, :] @ d) * dact(h)
    return norms / N

sigmoid_norms = measure(lambda a: 1 / (1 + np.exp(-a)), lambda h: h * (1 - h))

# Figures quoted by the two exercises below and their solutions,
# from the Python measurements.
stopifnot(isTRUE(all.equal(py$sigmoid_norms, py$norms)))
sci <- function(v, digits) {
  parts <- strsplit(formatC(v, format = "e", digits = digits), "e")[[1]]
  sprintf("%s \\times 10^{%d}", parts[1], as.integer(parts[2]))
}
summarize_norms <- function(norms) {
  k <- length(norms)
  list(
    first = norms[1],
    last = norms[k],
    ratio = norms[1] / norms[k],
    geomean = exp(mean(log(norms[-1] / norms[-k])))
  )
}
english_count <- function(n) {
  c("one", "two", "three", "four", "five", "six", "seven", "eight", "nine")[n]
}
sig <- summarize_norms(py$norms)
relu <- summarize_norms(py$relu_norms)

```

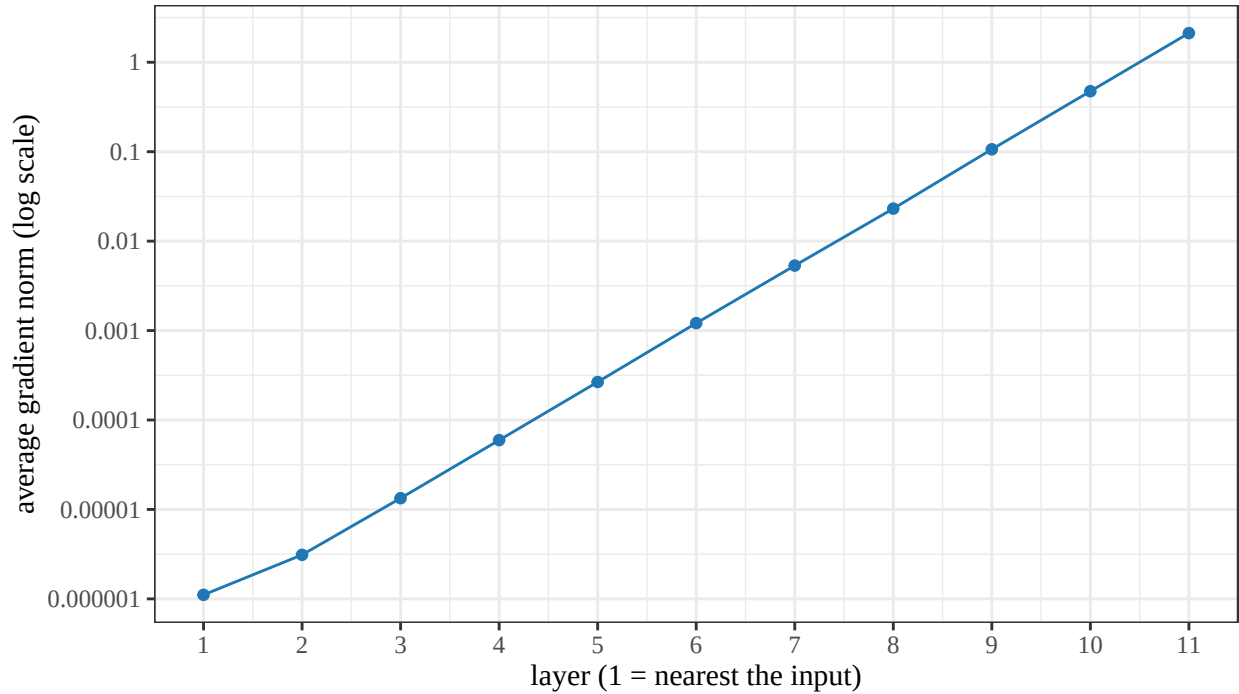


Figure 1: Frobenius norm of the gradient of the loss with respect to each layer’s weights, averaged over the 200 sigmoid networks of the measurement, from the layer nearest the input to the output layer. The vertical axis is logarithmic.

Exercise 8.1 (Find out what the recursion does to the size of a gradient). Exercise 6.1 says depth costs no extra arithmetic. It says nothing about whether the numbers coming back are useful when they arrive.

Unroll the recursion from the output down to layer 1:

$$\tilde{\delta}^{(1)} = \left[\prod_{k=2}^{K+1} (\text{a matrix, then an elementwise } f') \right] \tilde{\delta}^{(K+1)}.$$

Write that product out in full, as an alternating chain of matrices $W_{1:L_{k-1},:}^{(k)}$ and diagonal^a matrices holding $f'(a^{(k)})$. How many factors of f' does $\tilde{\delta}^{(1)}$ carry?

^a<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-diagonal-matrix>

Solution

Solution 8.1. Writing $F^{(k)} = \text{diag}(f'(a^{(k)}))$,

$$\tilde{\delta}^{(1)} = F^{(1)}(W_{1:L,:}^{(2)})F^{(2)}(W_{1:L,:}^{(3)}) \dots F^{(K)}(W_{1:L,:}^{(K+1)})\tilde{\delta}^{(K+1)},$$

so $\tilde{\delta}^{(1)}$ carries K factors of f' — one per hidden layer it has passed through. The count is the whole story: every one of them is a number no larger than expit’s maximum slope.

Exercise 8.2 (Find out what the recursion does to the size of a gradient (part 2)). Use the setup of Exercise 8.1.

For $f = \text{expit}$, Exercise 4.6 gives $\text{expit}'(a) = h(1 - h)$. Give its maximum over all a , and deduce a bound on how much each activation factor can multiply the size of $\tilde{\delta}$ by.

Solution

Solution 8.2. $\text{expit}'(a) = h(1 - h)$ with $h \in (0, 1)$, maximized at $h = 1/2$:

$$\begin{aligned}\max_a \text{expit}'(a) &= \frac{1}{2} \times \frac{1}{2} \\ &= \frac{1}{4}.\end{aligned}$$

Each activation factor therefore multiplies the size of $\tilde{\delta}$ by at most $1/4$, and by less than that whenever any unit is away from the middle of its range.

Exercise 8.3 (Find out what the recursion does to the size of a gradient (part 3)). Use the setup of Exercise 8.1.

Ignoring the weight matrices for a moment, evaluate that bound for $K = 2, 5, 10$ and 20 hidden layers. State plainly what it says about the gradient reaching layer 1 of a twenty-layer sigmoid^a network.

^a<https://morrison-lab.github.io/mds/functions.html#def-logistic-function>

Solution

Solution 8.3. The best case is 4^{-K} :

K	4^{-K}
2	6.25×10^{-2}
5	9.8×10^{-4}
10	9.5×10^{-7}
20	9.1×10^{-13}

At twenty layers the gradient reaching layer 1 is smaller than the one at the output by twelve orders of magnitude **in the most favorable case available** — every unit sitting exactly at $h = 1/2$, which is also the case where the units are carrying the least information. In double precision it is not yet zero; in the single precision used for training, and against the noise of a stochastic gradient, it is effectively zero.

Exercise 8.4 (Find out what the recursion does to the size of a gradient (part 4)). Use the setup of Exercise 8.1.

The bound is worst-case only if the weights do nothing. Say what a weight matrix could do to push in the opposite direction, and why that remedy conflicts with Exercise 4.6 in the other direction. Then say what goes wrong at the other extreme, when the weights are small: follow a signal forward through the chain rather than a gradient backward, and say what reaches the output.

Solution

Solution 8.4. Large weights. $W^{(k+1)}$ can multiply $\tilde{\delta}$ up, and with entries large enough it can exceed the factor of $1/4$.

But the same large weights drive the pre-activations $a^{(k)}$ to large magnitudes on the forward pass, which is precisely where expit' collapses: $\text{expit}'(4) = 0.018$, $\text{expit}'(6) = 0.0025$. So the remedy makes the problem worse, and by more than it helps. Small weights fail the other way, and the failure is on the forward pass rather than the backward one. A matrix with small entries shrinks $a^{(k)}$ towards 0, where expit is close to affine and every unit reports very nearly $1/2$ whatever its input^a was. After ten such layers the network has compressed the inputs^b into a small neighborhood of one point, so the output carries almost no information about x — and there is nothing useful for the gradient to be large *about*. The signal shrinks to nothing going forward; with large weights the gradient shrinks to nothing coming back.

The two failure modes are the same phenomenon seen from each end, which is why no choice of scale for

a sigmoid^c network escapes it — there is a best range of scales, it is narrow, and depth narrows it.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^b<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^c<https://morrison-lab.github.io/mds/functions.html#def-logistic-function>

Exercise 8.5 (Find out what the recursion does to the size of a gradient (part 5)). Use the setup of Exercise 8.1.

Section 8 measures exactly this, on 200 freshly initialized sigmoid^a networks with 10 hidden layers of 20 units, 20 inputs^b and 3 classes. Averaged over the 200, the Frobenius norm^c of $\partial\mathcal{L}/\partial W^{(k)}$ runs from 1.11×10^{-6} at layer 1 to 2.12 at the output — a ratio of 5.2×10^{-7} — and the ratio between consecutive layers has geometric mean^d 4.25. Say which of those numbers is the 1/4 of Exercise 8.2 reappearing, and then describe what training such a network looks like: which parameters move, which do not, and what the network effectively becomes.

^a<https://morrison-lab.github.io/mds/functions.html#def-logistic-function>

^b<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^c<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-frobenius-norm>

^d<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

Solution

Solution 8.5. The geometric mean^a of 4.25 is the 1/4 of Solution 8.2 reappearing, almost exactly: the bound said each layer costs at most a factor of four, and freshly initialized networks lose very nearly that full factor. The end-to-end ratio 5.2×10^{-7} adds no evidence of its own: the ten consecutive ratios multiply to it, so it is exactly the geometric mean to the power -10 . The evidence is the geometric mean itself, which is within 6% of the factor of four the bound allows, so nothing else has a significant effect — the weight matrices are neither restoring the gradient nor making it much worse.

Training such a network moves the last layer or two and leaves the rest where initialization put them. The network is then, in effect, **a random feature map with a trained classifier on top**: whatever the early layers compute, they compute it by accident, and the only thing being fitted is the last matrix. That is a real model. Fitting a linear classifier on top of a fixed random map is a method in its own right, and one with theory behind it — Rahimi and Recht’s random features, published in 2007, are the best-known version. But it is not the deep network anyone intended to train, and ten sigmoid^b layers are a very expensive way to draw random features.

^a<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

^b<https://morrison-lab.github.io/mds/functions.html#def-logistic-function>

Exercise 8.6 (Find out what the recursion does to the size of a gradient (part 6)). Use the setup of Exercise 8.1.

Does making the learning rate larger fix it? Answer for the early layers and the late layers separately.

Solution

Solution 8.6. No, and it fails in both directions at once.

A learning rate large enough to move the early layers is 10^6 times too large for the last layer, which will diverge immediately. A learning rate that keeps the last layer stable leaves the early layers unchanged. There is no single number that serves both. Fixing that needs either a per-layer step size — the direction the adaptive optimizers of gradient descent take — or an activation function that does not shrink the gradient at all. Exercise 8.7 takes the second route.

The diagnosis names the cause precisely enough to suggest a remedy: if $\max_a f'(a)$ is small, use a function whose derivative is not.

Exercise 8.7 (Change the activation function and see what moves). Exercise 8.1 traced the problem to a single number: the largest value expit' can take. That suggests a simple fix — use an activation function whose derivative is not small.

The rectified linear unit (ReLU) activation function is

$$f(a) = \max(a, 0).$$

Give $f'(a)$, and say what happens at $a = 0$.

Solution

Solution 8.7.

$$f'(a) = \begin{cases} 1 & a > 0 \\ 0 & a < 0. \end{cases}$$

At $a = 0$ the function is not differentiable: the left slope is 0 and the right slope is 1. Implementations pick one by convention, usually 0. This is less troubling than it sounds — the set of parameters for which any pre-activation falls exactly on 0 has measure zero — but it is worth knowing that the “gradient” your library returns is a **subgradient**, and that the convergence guarantees of gradient descent were not written for it.

Exercise 8.8 (Change the activation function and see what moves (part 2)). Use the setup of Exercise 8.7. Substitute your answer into the unrolled product of Exercise 8.1. What does each activation factor do to the size of $\tilde{\delta}$ now, and what replaces the 4^{-K} bound?

Solution

Solution 8.8. Each factor is now 1 or 0: the derivative either passes through unchanged or is switched off entirely.

The product of K factors is therefore 1 along any path whose units are all active, rather than 4^{-K} . There is no shrinkage bound to replace it, because there is no shrinkage: the activation function selects which paths carry gradient and does not attenuate the gradient on the paths it selects.

Exercise 8.9 (Change the activation function and see what moves (part 3)). Use the setup of Exercise 8.7. Repeating Section 8 with ReLU in place of expit gives a first-to-last gradient ratio of 5.6×10^{-2} where the sigmoid^a gave 5.2×10^{-7} , and a geometric-mean per-layer ratio of 1.33 against 4.25. Say which of those four numbers is the one the fix was aimed at, and whether the remaining decay is the same phenomenon.

^a<https://morrison-lab.github.io/mds/functions.html#def-logistic-function>

Solution

Solution 8.9. The ReLU measurement is the Python one with the sigmoid^a’s two functions swapped for ReLU’s:

```
# ReLU in place of the sigmoid: max(a, 0), with derivative [h > 0].
# Where it runs, before "Change the activation function and see what moves",
# this line is hidden, because it gives away part 1 of that exercise, and the
# exercise quotes its result, so it cannot run later. Its solution shows it.
relu_norms = measure(lambda a: np.maximum(a, 0), lambda h: (h > 0) * 1.0)
```

The fix was aimed at the **per-layer ratio**: 4.25 was the 1/4 of expit' made visible, and 1.33 is the ratio once that factor is gone.

The remaining decay is a different phenomenon. It comes from the weight matrices and from the units

that happen to be inactive on a given input^b, not from an upper limit on f' — so it depends on the initialization scale and can be tuned, where the sigmoid's 1/4 could not be. Over ten layers the difference is 5.6×10^{-2} against 5.2×10^{-7} : five orders of magnitude, from one change to one function.

^a<https://morrison-lab.github.io/mds/functions.html#def-logistic-function>

^b<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Exercise 8.10 (Change the activation function and see what moves (part 4)). Use the setup of Exercise 8.7.

The fix has a cost. Consider a unit whose pre-activation is negative for **every** training example. What is its gradient, what will it be at the next step, and what have you lost?

Solution

Solution 8.10. Its gradient is zero, because $f'(a) = 0$ for every example, so $\delta_j^a = 0$ and none of the weights feeding it move. Since they do not move, the pre-activation is negative for every example at the next step too, and at every step after that.

The unit is **dead**: it has no output and no prospect of recovering one, and it will consume memory and arithmetic for the rest of training without contributing anything. You have lost the guarantee, which the sigmoid^a did give, that every parameter receives some gradient from every example. The leaky variants — $f(a) = \max(a, \alpha a)$ for a small $\alpha > 0$, which Maas, Hannun and Ng introduced in 2013 — exist for exactly this reason: a negative pre-activation still passes a small gradient, so no unit can be permanently stuck. They give up the exact zeros that make ReLU fast.

^a<https://morrison-lab.github.io/mds/functions.html#def-logistic-function>

Exercise 8.11 (Change the activation function and see what moves (part 5)). Use the setup of Exercise 8.7.

ReLU is linear on each side of the origin, and the neural networks chapter showed that a linear f collapses the network to a single matrix. Explain why this argument does not apply, and check your explanation by building the exclusive-or (XOR) function from two ReLU units.

Solution

Solution 8.11. Because ReLU is piecewise linear rather than linear. The neural networks chapter's collapse needed one affine map valid everywhere; ReLU gives a different affine map on each side of the hyperplane $w_j \cdot x = 0$, and with L units the weights can cut the plane into as many as $1 + L + \binom{L}{2}$ pieces, with a different affine function on each.

XOR in two units, on binary inputs^a:

$$h_1 = \max(x_1 + x_2, 0), \quad h_2 = \max(x_1 + x_2 - 1, 0), \quad y = h_1 - 2h_2.$$

x_1	x_2	h_1	h_2	y	XOR
0	0	0	0	0	0
0	1	1	0	1	1
1	0	1	0	1	1
1	1	2	1	0	0

Both hidden units see the same quantity $x_1 + x_2$ and differ only in their bias^b; the second one switches on only when both inputs are, and subtracting twice it turns the ramp back down.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^b<https://morrison-lab.github.io/sds/estimation.html#def-bias>

Exercise 8.12 (Change the activation function and see what moves (part 6)). Use the setup of Exercise 8.7.

The construction in Exercise 8.11 is exact, where the sigmoid^a version in the neural networks chapter was only accurate to about 4.5×10^{-5} . Say which property of ReLU gives that, and what ReLU gives up in exchange.

^a<https://morrison-lab.github.io/mds/functions.html#def-logistic-function>

Solution

Solution 8.12. Exactness comes from ReLU being **unbounded above and exactly zero below**. The sigmoid^a can only approach 0 and 1 asymptotically, so its logic gates are always off by about $\text{expit}(-10) = 4.5 \times 10^{-5}$ at each unit, and no choice of finite weights removes that fixed error. The error does not grow with depth — weights of ± 20 push each pre-activation about 10 away from the boundary, so every stage re-sharpens its input^b and the final output of the sigmoid network is off by 4.5440×10^{-5} against the 4.5398×10^{-5} of a single unit. The error has a fixed size, and the ReLU construction has none.

ReLU gives up the probabilistic reading. A sigmoid output is in $(0, 1)$ and we can report it as a probability. A ReLU output is any non-negative number and means^c nothing on its own. So we use ReLU for hidden layers and the output layer keeps $g = \text{softmax}$ or $g = \text{expit}$: the network computes in unbounded units and converts to probabilities once, at the end.

^a<https://morrison-lab.github.io/mds/functions.html#def-logistic-function>

^b<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^c<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

i Watch

Hutchinson covers exploding and vanishing gradients in Deep Neural Networks^a (21 min) [3].

^ahttps://facultyweb.cs.wvu.edu/~hutchib2/video_lectures/data371/#dnns

i Read

GeeksforGeeks explains:

- vanishing and exploding gradients^a [13]
- the ReLU activation function^b [14]

^a<https://www.geeksforgeeks.org/deep-learning/vanishing-and-exploding-gradients-problems-in-deep-learning/>

^b<https://www.geeksforgeeks.org/deep-learning/relu-activation-function-in-deep-learning/>

9 Exploding gradients

ReLU took away the $1/4$ cap on each factor of f' . The weights now decide each factor's size.

Exercise 9.1 (Turn the weights up). Exercise 8.7 replaced a factor of f' that was at most $1/4$ with one that is 0 or 1. That leaves the weights alone to set how much each layer of the chain in the vanishing-gradient exercise (Exercise 8.1) multiplies a vector by, and nothing stops that from being more than one.

Draw every weight of layer k from $\mathcal{N}(0, s^2/(L_{k-1} + 1))$, with ReLU units throughout. The measurement of Section 8 used $s = 1$.

Take a layer with input^a $\tilde{h}$, ignore its bias^b, and write n for its fan-in. Give the mean^c and variance^d of one pre-activation a_j , and say what ReLU does to it. Deduce that, when the layer is as wide as its input, it multiplies the typical size of $\tilde{h}$ by about $s/\sqrt{2}$. Argue that the backward pass multiplies $\tilde{\delta}$ by

the same factor, and say which s makes that factor 1.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^b<https://morrison-lab.github.io/sds/estimation.html#def-bias>

^c<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

^d<https://morrison-lab.github.io/pds/variance-covariance.html#def-variance>

Solution

Solution 9.1. Without the bias, $a_j = \sum_{i=1}^n w_{ij}h_i$, a sum of independent terms with mean 0. Taking the weight variance as s^2/n (the +1 added to the fan-in is for the bias, set aside here),

$$\begin{aligned}\mathbb{E}[a_j] &= 0, \\ \text{Var}(a_j) &= \frac{s^2}{n} \sum_{i=1}^n h_i^2 \\ &= \frac{s^2 \|\tilde{h}\|^2}{n}.\end{aligned}$$

ReLU keeps a_j when it is positive and replaces it with 0 otherwise. Since a_j is symmetric about 0, that keeps half its mean square: $\mathbb{E}[\max(a_j, 0)^2] = s^2 \|\tilde{h}\|^2 / (2n)$. Adding up the n units of a layer as wide as its input, the output has mean square $s^2 \|\tilde{h}\|^2 / 2$, so the layer multiplies the typical size of $\tilde{h}$ by $s/\sqrt{2}$.

The backward pass has the same two ingredients: $\tilde{\delta}^{(k)} = F^{(k)} W_{1:L,:}^{(k+1)} \tilde{\delta}^{(k+1)}$ multiplies by a matrix of the same independent weights, then by $F^{(k)}$, whose diagonal is 1 for the units that are on and 0 for the rest. Treating $\tilde{\delta}^{(k+1)}$ as independent of $W^{(k+1)}$, and about half the units as on, as He and colleagues do [15, Sec. 2.2], the same count gives the same factor, $s/\sqrt{2}$. Going backward the bias row is dropped, so the factor there is in fact $\frac{s}{\sqrt{2}} \sqrt{L/(L+1)}$, just under 1 at $s = \sqrt{2}$: part of why the gradient at $s = \sqrt{2}$ still falls a little towards layer 1.

So $s = \sqrt{2}$ makes it 1: weights with variance $2/n$.

Exercise 9.2 (Turn the weights up (part 2)). Use the setup of Exercise 9.1.

With $K = 10$ hidden layers, by what factor does Exercise 9.1 predict $\|\tilde{h}^{(K)}\|$ to differ from $\|x\|$ at $s = 1$, $\sqrt{2}$ and 2?

Solution

Solution 9.2. The factor $(s/\sqrt{2})^K$ is $2^{-5} = 1/32$ at $s = 1$, exactly 1 at $s = \sqrt{2}$, and $2^5 = 32$ at $s = 2$.

Exercise 9.3 (Turn the weights up (part 3)). Use the setup of Exercise 9.1.

Repeating Section 8 with ReLU units at those three values of s gives these averages over the same 200 networks:

	$s = 1$	$s = \sqrt{2}$	$s = 2$
$\ x\ $	4.44	4.44	4.44
$\ \tilde{h}^{(K)}\ $	0.925	3.80	87.1
$\ \partial \mathcal{L} / \partial W^{(1)}\ $	0.0629	2.21	91.7
$\ \partial \mathcal{L} / \partial W^{(K+1)}\ $	1.13	3.59	91.6

Compare the second row with Exercise 9.2, and say what keeps the $s = 1$ figure from falling as far as predicted. Then explain why, at $s = 2$, the gradient is about the same size at layer 1 as at the output, although everything in the network is growing.

Solution

Solution 9.3. The measurement uses the same draws as Section 8. At $s = 1$ it reproduces the ReLU norms^a of Exercise 8.7, which the page checks before quoting any of these figures.

```
# The measurement with ReLU units, with every weight's standard
# deviation multiplied by `scale`. Besides the per-layer gradient norms it
# returns the average norm and mean square of each layer's output, the norm
# of each network's whole gradient, and the norm of each network's  $W^{(1)}$ .
def measure_scaled(scale, K=10, L=20, D=20, C=3, N=200, seed=0):
    rng = np.random.default_rng(seed)
    widths = [D] + [L] * K + [C]
    norms = np.zeros(K + 1)
    forward = np.zeros(K + 1)
    forward_sq = np.zeros(K + 1)
    total = np.zeros(N)
    first_weights = np.zeros(N)
    for i in range(N):
        W = [rng.normal(0, scale / np.sqrt(widths[k] + 1),
                        (widths[k] + 1, widths[k + 1]))
              for k in range(K + 1)]
        post = [np.concatenate(([1.0], rng.normal(0, 1, D)))]
        pre = []
        for k in range(K + 1):
            a = W[k].T @ post[k]
            pre.append(a)
            if k < K:
                post.append(np.concatenate(([1.0], np.maximum(a, 0))))
        sizes = np.array([np.linalg.norm(p[1:]) for p in post])
        forward += sizes
        forward_sq += sizes ** 2
        e = np.exp(pre[K] - pre[K].max())
        t = np.zeros(C)
        t[rng.integers(C)] = 1.0
        d = e / e.sum() - t
        sq = 0.0
        for k in range(K, -1, -1):
            g = np.linalg.norm(np.outer(post[k], d))
            norms[k] += g
            sq += g ** 2
            if k > 0:
                h = post[k][1:]
                d = (W[k][1:, :] @ d) * (h > 0)
        total[i] = np.sqrt(sq)
        first_weights[i] = np.linalg.norm(W[0])
    return {"norms": norms / N, "forward": forward / N,
            "forward_sq": forward_sq / N, "total": total,
            "first_weights": first_weights}

scaled = {name: measure_scaled(scale) for name, scale in
          [("one", 1.0), ("root2", np.sqrt(2)), ("two", 2.0)]}
```

The ordering is right, with $s = \sqrt{2}$ holding the size steady and $s = 2$ growing it by a factor of 19.6. But $s = 1$ shrinks it by a factor of only 4.80, where Solution 9.2 predicted 32.

Solution 9.1 left out the bias^b. Every layer's input^c is $[1; \tilde{h}]$, so every pre-activation keeps a term with

variance^d $s^2/(L+1)$ however small $\tilde{h}$ gets. With it, the mean^e square $m_k = \mathbb{E}\|\tilde{h}^{(k)}\|^2$ obeys

$$m_k = r(m_{k-1} + 1), \quad r = \frac{Ls^2}{2(L+1)}.$$

At $s = 1$, $r = \frac{10}{21}$, and m_k settles at the fixed point $r/(1-r) = \frac{10}{11}$, a norm^f of about 0.953, which stops the $s = 1$ network.

Solution 9.1 also predicts mean squares, where the table averages norms. Run from the measured mean square of $\tilde{x}$, the recursion predicts root-mean-square norms for $\tilde{h}^{(K)}$ of 0.960, 4.49 and 119; the networks give 0.963, 4.55 and 115. The average norm at $s = 2$, 87.1, is smaller than its root mean square because the growth varies from network to network, and the networks that grow fastest raise a mean square more than a mean.

The gradient at layer k is $\partial\mathcal{L}/\partial W^{(k)} = [1; \tilde{h}^{(k-1)}](\tilde{\delta}^{(k)})^\top$. Going forward, $\tilde{h}$ gains a factor of $g = s/\sqrt{2}$ per layer; going backward, $\tilde{\delta}$ gains the same g per layer. So $\tilde{h}^{(k-1)}$ carries g^{k-1} and $\tilde{\delta}^{(k)}$ carries g^{K+1-k} , and their product carries $\tilde{g}^K$ at every layer. So the $s = 2$ gradient is 91.7 at layer 1 and 91.6 at the output: the growth is the same everywhere, and large. At $s = 1$ the bias stops $\tilde{h}$ from shrinking but nothing stops $\tilde{\delta}$, so the gradient there still falls towards layer 1.

^a<https://morrison-lab.github.io/mds/linear-algebra-inner-products.html#def-norm>

^b<https://morrison-lab.github.io/sds/estimation.html#def-bias>

^c<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^d<https://morrison-lab.github.io/pds/variance-covariance.html#def-variance>

^e<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

^f<https://morrison-lab.github.io/mds/linear-algebra-inner-products.html#def-norm>

Exercise 9.4 (Turn the weights up (part 4)). Use the setup of Exercise 9.1.

At $s = 2$, $\|W^{(1)}\|$ averages 8.96. Gradient descent^a with learning rate $\alpha = 0.1$ would move $W^{(1)}$ by $\alpha\|\partial\mathcal{L}/\partial W^{(1)}\|$ on its first step. Compare the two, and say why the gradient that chose that step says little about where the step lands.

^a<https://morrison-lab.github.io/mds/optimization.html#def-gradient-descent>

Solution

Solution 9.4. The step has length about $0.1 \times 91.7 = 9.17$, against the norm $\|W^{(1)}\| \approx 8.96$: one step changes $W^{(1)}$ by about as much as $W^{(1)}$ is. The gradient describes the loss only near the current weights, as the slope^a of its linear approximation there. A step that size lands where the approximation no longer holds: units switch on and off, the loss can rise instead of fall, and a few such steps in a row can overflow to Inf and then NaN.

^a<https://morrison-lab.github.io/mds/algebra-exponentials.html#def-slope>

Definition 9.1 (Exploding gradient). An **exploding gradient** is one whose size grows geometrically with the number of layers it passes back through, because the factors of the chain in Exercise 8.1 are larger than one. A gradient step that large overshoots the region where the gradient describes the loss [16].

Definition 9.2 (He initialization). **He initialization** draws each weight feeding a layer of ReLU units from $\mathcal{N}(0, 2/n)$, where n is the layer's fan-in. It keeps the mean square of the vector passing through a layer as wide as its input unchanged, on the forward pass and on the backward pass [15, Sec. 2.2]. He and colleagues set the biases to zero. The experiment in Exercise 9.1 draws them too, and its $s = \sqrt{2}$ is the same rule with the bias counted in the fan-in.

Definition 9.3 (Glorot initialization). **Glorot initialization** takes the scale of a layer’s random weights from the layer’s shape rather than from experiment [17]. For a layer with n_{in} inputs and n_{out} outputs, draw each weight with mean zero and

$$\text{Var}(w) = \frac{2}{n_{\text{in}} + n_{\text{out}}},$$

which is a standard deviation of $\sqrt{2/(n_{\text{in}} + n_{\text{out}})}$ for a normal draw, or bounds $\pm\sqrt{6/(n_{\text{in}} + n_{\text{out}})}$ for the uniform draw the original paper states.

Example 9.1 (A layer from 784 inputs to 256 units). Take a layer with $n_{\text{in}} = 784$ inputs, the pixels of a 28×28 image, and $n_{\text{out}} = 256$ units.

```
# The Glorot variance, the standard deviation of the normal version,
# and the half-width of the uniform version, which has the same variance.
n_in <- 784
n_out <- 256
glorot_var <- 2 / (n_in + n_out)
glorot_sd <- sqrt(glorot_var)
glorot_bound <- sqrt(6 / (n_in + n_out))
# The variance He initialization would use, which counts only the inputs.
he_var <- 2 / n_in
# A uniform distribution on (-a, a) has variance a^2 / 3,
# so this equals glorot_var; it confirms the two versions match.
stopifnot(isTRUE(all.equal(glorot_bound^2 / 3, glorot_var)))
```

Glorot initialization draws each of the 784×256 weights with variance $2/(784 + 256) = 0.00192$, so a normal draw has standard deviation 0.0439, and a uniform draw is on $(-a, a)$ with $a = 0.076$. He initialization, which counts only the inputs, would use variance $2/784 = 0.00255$. Glorot’s is smaller, because its denominator counts the 256 outputs as well as the inputs.

The denominator is a compromise between two requirements that cannot both be met unless the layer is square. Holding the variance of the activations steady from one layer to the next asks for $1/n_{\text{in}}$; holding the variance of the back-propagated error steady asks for $1/n_{\text{out}}$. The Glorot variance is the harmonic mean of those two. [He initialization](#) makes the same trade for ReLU units, but keeps only the forward requirement, with the factor of 2 that the rectifier’s discarded half calls for.

The two schemes set the scale of the weights. Biases are a separate and less important choice, because a bias belongs to one unit and so breaks no symmetry by itself. Zero is the usual value. A common exception starts the biases of rectified linear units at a small positive number, 0.1 say, so that the units begin on the active side of the rectifier where the derivative is not zero.

Watch

Hutchinson covers weight initialization^a (16 min) [3].

^ahttps://facultyweb.cs.wvu.edu/~hutchib2/video_lectures/data371/#weight_initialization

Read

GeeksforGeeks explains weight initialization techniques^a [18].

^a<https://www.geeksforgeeks.org/machine-learning/weight-initialization-techniques-for-deep-neural-networks/>

Exercise 9.5 (Cap the length of a step). He initialization (Definition 9.2) chooses a good starting point, but training moves the weights away from it, and a gradient can still grow too large later on. A

safeguard should touch only the steps that would do damage.

Stack the gradient of every parameter into one vector g , and pick a threshold $\tau > 0$.

Write a rule that returns a vector pointing the same way as $\tilde{g}$, returns $\tilde{g}$ itself when $\|\tilde{g}\| \leq \tau$, and otherwise returns a vector of length τ .

Solution

Solution 9.5.

$$\tilde{g} \leftarrow \min\left(1, \frac{\tau}{\|\tilde{g}\|}\right) \tilde{g}.$$

The multiplier is positive, so the direction does not change. It is 1 when $\|\tilde{g}\| \leq \tau$, and otherwise it is $\tau/\|\tilde{g}\|$, which makes the length exactly τ .

Exercise 9.6 (Cap the length of a step (part 2)). Use the setup of Exercise 9.5.

On the networks of Exercise 9.1, the norm^a of g has median 1.50, 6.64 and 216 at $s = 1$, $\sqrt{2}$ and 2, and with $\tau = 10$ your rule would change 0%, 34% and 76% of them. With $\alpha = 0.1$, how long can a step now be?

^a<https://morrison-lab.github.io/mds/linear-algebra-inner-products.html#def-norm>

Solution

Solution 9.6. A step is now at most $\alpha\tau = 1$ long.

Exercise 9.7 (Cap the length of a step (part 3)). Use the setup of Exercise 9.5.

How does that length compare with the step length of Exercise 9.4?

Solution

Solution 9.7. That compares with the 9.17 of Solution 9.4 of Exercise 9.4, which was about the size of $W^{(1)}$ itself.

Exercise 9.8 (Cap the length of a step (part 4)). Use the setup of Exercise 9.5.

What do the three percentages of Exercise 9.6 say about when the rule is the right remedy?

Solution

Solution 9.8. At $s = 1$ the rule never acts, and at $s = 2$ it acts on most networks. At $s = \sqrt{2}$ it acts on 34%, which says $\tau = 10$ is low for those networks. We choose τ from the gradient norms^a that a healthy network actually produces, for example their average over many updates [19, Sec. 3.2]. The rule is a guard against the occasional spike. A network that needs it on most steps is badly scaled, and the remedy is to fix the scale, starting with Definition 9.2.

^a<https://morrison-lab.github.io/mds/linear-algebra-inner-products.html#def-norm>

Exercise 9.9 (Cap the length of a step (part 5)). Use the setup of Exercise 9.5.

Explain why the rule cannot help with the vanishing gradients of Exercise 8.1.

Solution

Solution 9.9. The rule only ever shrinks a gradient, and it multiplies every layer's part of g by the same factor. A vanishing gradient is too small, and its trouble is the ratio between layer 1 and the output, which the rule leaves exactly as it was.

Definition 9.4 (Gradient clipping). **Gradient clipping** by norm replaces the gradient $\tilde{g}$ with $\min(1, \tau/\|\tilde{g}\|)\tilde{g}$ before each update, for a chosen threshold τ [16], [19, Sec. 3.2].

Example 9.2 (Clipping a large gradient). Fit mpg on standardized horsepower in the Auto data [20] by gradient descent from $\vec{\beta} = \vec{0}$. At the start the gradient of the sample mean squared error (sMSE) is large, because the fitted line is far from the data. Compute it, and clip it to norm $\tau = 5$:

9.1 Python

```
import numpy as np
import pandas as pd

auto_gc = pd.read_csv("data/auto.csv")[["mpg", "horsepower"]].dropna()
y_gc = auto_gc["mpg"].to_numpy(float)
hp_gc = auto_gc["horsepower"]
X_gc = np.column_stack([np.ones(len(y_gc)), ((hp_gc - hp_gc.mean()) / hp_gc.std()).to_numpy()])

# Gradient of the mean squared error at beta = 0.
g_gc = 2 * X_gc.T @ (X_gc @ np.zeros(2) - y_gc) / len(y_gc)

# Clip by norm: shrink the gradient to length clip_gc if it is longer, keep its direction.
clip_gc = 5
g_clipped_gc = min(1, clip_gc / np.linalg.norm(g_gc)) * g_gc
print(g_gc, np.linalg.norm(g_gc))

[-46.89183673  12.12025566] 48.4328911950955
print(g_clipped_gc, np.linalg.norm(g_clipped_gc))

[-4.84090827  1.25124222] 5.0
```

9.2 R

```
auto_gc <- na.omit(read.csv("data/auto.csv")[, c("mpg", "horsepower")])
y_gc <- auto_gc$mpg
X_gc <- cbind(1, as.vector(scale(auto_gc$horsepower)))

# Gradient of the mean squared error at beta = 0.
g_gc <- as.vector(2 * t(X_gc) %*% (X_gc %*% c(0, 0) - y_gc) / length(y_gc))

# Clip by norm: shrink the gradient to length clip_gc if it is longer, keep its direction.
clip_gc <- 5
g_clipped_gc <- min(1, clip_gc / sqrt(sum(g_gc^2))) * g_gc
rbind(gradient = g_gc, clipped = g_clipped_gc)

      [,1]      [,2]
gradient -46.891837 12.120256
clipped  -4.840908  1.251242
```

```
c(norm = sqrt(sum(g_gc^2)), clipped_norm = sqrt(sum(g_clipped_gc^2)))
```

```
norm clipped_norm
48.43289      5.00000
```

The gradient has norm 48.4, far above the threshold, so clipping multiplies it by 0.103. The clipped gradient has norm 5.0 and points the same way. Clipping changes how far a step goes and not which way it goes. Once the fit improves and the gradient norm falls below 5, clipping leaves it alone.

10 Choosing the hidden non-linearity

expit and ReLU are two of several functions in use for hidden units. The measurement can compare them too.

Exercise 10.1 (Compare four non-linearities). This chapter has used expit and ReLU as the non-linearity f , also called the activation function. Two more are common in hidden layers: tanh, which part 5 of What the logistic function does showed is $\tanh(a) = 2 \expit(2a) - 1$, and

$$f(a) = \max(a, \alpha a) \quad \text{for a small } \alpha > 0,$$

called the **leaky ReLU**; take $\alpha = 0.01$.

Use $\tanh(a) = 2 \expit(2a) - 1$ to show that

$$\begin{aligned} \tanh'(a) &= 4 \expit'(2a) \\ &= 1 - \tanh(a)^2. \end{aligned}$$

Give the largest value $\tanh'$ takes, and compare it with the $1/4$ that Exercise 8.2 found for $\expit'$.

Solution

Solution 10.1. Differentiating $\tanh(a) = 2 \expit(2a) - 1$ gives

$$\begin{aligned} \tanh'(a) &= 2 \cdot 2 \expit'(2a) \\ &= 4 \expit'(2a). \end{aligned}$$

Since $\expit' = \expit(1 - \expit)$ and $\expit(2a) = (1 + \tanh a)/2$,

$$\begin{aligned} \tanh'(a) &= 4 \cdot \frac{1 + \tanh a}{2} \cdot \frac{1 - \tanh a}{2} \\ &= 1 - \tanh(a)^2. \end{aligned}$$

Its largest value is $4 \times \frac{1}{4} = 1$, at $a = 0$: four times $\expit'$'s. A factor of the chain in Exercise 8.1 can now be as large as 1, so the bound no longer forces the gradient to shrink.

Exercise 10.2 (Compare four non-linearities (part 2)). Use the setup of Exercise 10.1.

A unit is **saturated** when its derivative is close to 0, so that almost no gradient passes through it. The derivatives at $a = 0, 2$ and 5 are:

	$a = 0$	$a = 2$	$a = 5$
$\expit'(a)$	0.250	0.105	0.00665
$\tanh'(a)$	1.00	0.0707	0.000182

For each of the four functions, say for which a a unit is saturated, and what happens to a unit whose pre-activation is negative for every training example.

Solution

Solution 10.2. expit and tanh are saturated when $|a|$ is large, in either direction, and tanh more sharply: at $a = 5$ its derivative is 0.000182 against expit's 0.00665. A unit of either whose pre-activation is negative on every example still has a nonzero derivative there, so it keeps learning, slowly if it sits far out in the tail.

ReLU's derivative is exactly 1 for $a > 0$, so it is never saturated there, and exactly 0 for $a < 0$: by the definition, saturated completely. The leaky ReLU's derivative is α for $a < 0$: nearly saturated, but not completely. That difference matters for a unit negative on every example. The ReLU unit is dead, with a gradient of 0 that stays 0 (Solution 10.4 of Exercise 8.7); the leaky one still receives α of the gradient and can recover.

Exercise 10.3 (Compare four non-linearities (part 3)). Use the setup of Exercise 10.1.

Repeating Section 8 with each function in turn gives the ratio of the gradient at layer 1 to the gradient at the output, the root mean^a square (RMS) of the hidden units' derivatives, and the share of hidden units whose derivative is below 0.1:

	expit	tanh	ReLU	leaky ReLU
layer 1 / output	5.2×10^{-7}	0.269	0.0557	0.0551
RMS derivative	0.232	0.846	0.708	0.708
below 0.1	0.4%	0.7%	49.8%	49.9%

Explain why tanh keeps far more of the gradient than expit, although both can saturate. Explain ReLU's RMS derivative, given that about half its units pass nothing, and say whether this comparison is fair to ReLU. Then explain why the leaky ReLU's figures are almost those of ReLU, and when you would expect the two to differ.

Solution

Solution 10.3. These figures come from the measurement of Section 8, run once for each function on the same draws.

```

# The forward pass of the measurement above, on the same draws, recording
# every hidden unit's derivative. It returns their root mean square and the
# fraction of them below flat_cut; measure() supplies the gradient norms.
flat_cut = 0.1

def hidden_slopes(act, dact, K=10, L=20, D=20, C=3, N=200, seed=0):
    rng = np.random.default_rng(seed)
    widths = [D] + [L] * K + [C]
    slopes = []
    for _ in range(N):
        W = [rng.normal(0, 1 / np.sqrt(widths[k] + 1),
                        (widths[k] + 1, widths[k + 1]))
              for k in range(K + 1)]
        post = [np.concatenate(([1.0], rng.normal(0, 1, D)))]
        for k in range(K):
            h = act(W[k].T @ post[k])
            slopes.append(dact(h))
            post.append(np.concatenate(([1.0], h)))
        rng.integers(C)
    slopes = np.concatenate(slopes)
    return np.sqrt(np.mean(slopes ** 2)), np.mean(slopes < flat_cut)

leak = 0.01
nonlinearities = {
    "sigmoid": (lambda a: 1 / (1 + np.exp(-a)), lambda h: h * (1 - h)),
    "tanh": (np.tanh, lambda h: 1 - h ** 2),
    "relu": (lambda a: np.maximum(a, 0), lambda h: (h > 0) * 1.0),
    "leaky": (lambda a: np.maximum(a, leak * a),
              lambda h: np.where(h > 0, 1.0, leak)),
}
activation_results = {}
for name, (act, dact) in nonlinearities.items():
    rms, flat = hidden_slopes(act, dact)
    activation_results[name] = {"norms": measure(act, dact),
                              "rms": rms, "flat": flat}

```

Each layer multiplies $\tilde{\delta}$ by $F^{(k)}$, the diagonal^a matrix of its units' derivatives, so $\|F^{(k)}\tilde{\delta}\|^2 = \sum_j f'(a_j)^2 \delta_j^2$. With the derivatives independent^b of $\tilde{\delta}$, that multiplies the size of $\tilde{\delta}$ by about the root mean^c square of the derivatives, not their mean.

At this initialization the pre-activations are small, and almost no unit of expit or tanh is saturated (0.4% and 0.7%). Near $a = 0$, though, expit' is about 1/4 and tanh' is about 1: expit loses most of the gradient at every layer even unsaturated, and tanh keeps most of it.

On each input^d about half of ReLU's units are on, with derivative 1, and half are off, with derivative 0. The mean square of the derivatives is 1/2, so the root mean square is $1/\sqrt{2} \approx 0.707$: the $s/\sqrt{2}$ of Solution 10.1 of Exercise 9.1 at $s = 1$. The comparison is not fair to ReLU. The experiment draws weights with variance^e $1/(L + 1) = 1/21$, close to Glorot and Bengio's $2/(L_{k-1} + L_k) = 1/20$ for these equal widths, which is the scale recommended for tanh [17, Sec. 4.2]. For ReLU it is half of He initialization's variance (Definition 9.2). At He's scale, $s = \sqrt{2}$, the ReLU networks of Exercise 9.1 keep 0.614 of the gradient between the output and layer 1, against tanh's 0.269 here.

The leaky ReLU differs from ReLU only through the units that are off. At initialization each of those is off for one input, not dead, and α adds only 1% of the gradient through it, so the figures barely move. The two differ during training, when a unit can end up negative on every example: the ReLU unit stops learning, and the leaky one does not.

^a<https://morrison-lab.github.io/mds/linear-algebra-special-matrices.html#def-diagonal-matrix>
^b<https://morrison-lab.github.io/pds/independence.html#def-indpt>
^c<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>
^d<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>
^e<https://morrison-lab.github.io/pds/variance-covariance.html#def-variance>

Exercise 10.4 (Compare four non-linearities (part 4)). Use the setup of Exercise 10.1. Which would you use in the hidden layers of a deep network, and with which initialization?

Solution

Solution 10.4. ReLU is the usual default for the hidden layers of a deep network [21, Sec. 6.3], with [He initialization](#) [15, Sec. 2.2]: its derivative is exactly 1 wherever a unit is on, it is never saturated on that side, and it is cheap to compute. The leaky ReLU is the fix when too many units die. Of the two functions that saturate on both sides, tanh is the better hidden unit [21, Sec. 6.3.2], with Glorot and Bengio's initialization. expit is now rarely used in hidden layers, and belongs at the output of a binary classifier.

Definition 10.1 (Saturated unit). A unit is **saturated** when its pre-activation lies where the derivative of its non-linearity is close to 0, so that almost no gradient passes back through it. expit and tanh saturate when $|a|$ is large; ReLU is saturated completely for $a < 0$, where its derivative is 0.

Example 10.1 (How much gradient passes through a unit). The derivative of a non-linearity is the factor that multiplies the gradient on its way back through the unit. Compute it for the sigmoid, tanh and ReLU at several pre-activations a :

10.1 Python

```
import numpy as np

a_sat = np.array([-10.0, -5.0, -2.0, 0.0, 2.0, 5.0, 10.0])
sigmoid_sat = 1 / (1 + np.exp(-a_sat))

# Closed forms: sigmoid' = s(1 - s), tanh' = 1 - tanh^2, ReLU' = 1 for a > 0 and 0 otherwise.
d_sigmoid_sat = sigmoid_sat * (1 - sigmoid_sat)
d_tanh_sat = 1 - np.tanh(a_sat) ** 2
d_relu_sat = (a_sat > 0).astype(float)
print(np.column_stack([a_sat, d_sigmoid_sat, d_tanh_sat, d_relu_sat]).round(5))
```

```
[[-1.0000e+01  5.0000e-05  0.0000e+00  0.0000e+00]
 [-5.0000e+00  6.6500e-03  1.8000e-04  0.0000e+00]
 [-2.0000e+00  1.0499e-01  7.0650e-02  0.0000e+00]
 [ 0.0000e+00  2.5000e-01  1.0000e+00  0.0000e+00]
 [ 2.0000e+00  1.0499e-01  7.0650e-02  1.0000e+00]
 [ 5.0000e+00  6.6500e-03  1.8000e-04  1.0000e+00]
 [ 1.0000e+01  5.0000e-05  0.0000e+00  1.0000e+00]]
```

10.2 R

```

a_sat <- c(-10, -5, -2, 0, 2, 5, 10)
sigmoid_sat <- 1 / (1 + exp(-a_sat))

# Closed forms: sigmoid' = s(1 - s), tanh' = 1 - tanh^2, ReLU' = 1 for a > 0 and 0 otherwise.
round(cbind(a = a_sat,
            sigmoid = sigmoid_sat * (1 - sigmoid_sat),
            tanh = 1 - tanh(a_sat)^2,
            relu = as.numeric(a_sat > 0)), 5)

```

	a	sigmoid	tanh	relu
[1,]	-10	0.00005	0.00000	0
[2,]	-5	0.00665	0.00018	0
[3,]	-2	0.10499	0.07065	0
[4,]	0	0.25000	1.00000	0
[5,]	2	0.10499	0.07065	1
[6,]	5	0.00665	0.00018	1
[7,]	10	0.00005	0.00000	1

At $a = 0$ the sigmoid passes a quarter of the gradient and tanh passes all of it. By $|a| = 5$ the sigmoid passes 0.0066 and tanh passes 0.00018, so a unit whose pre-activation sits there is saturated: almost nothing flows back, and the weights feeding it barely learn. ReLU never shrinks the gradient for $a > 0$, but for every $a < 0$ the factor is exactly 0.

Definition 10.2 (Leaky ReLU (leaky rectifier)). The **leaky ReLU**, or **leaky rectifier**, is $f(a) = \max(a, \alpha a)$ for a small $\alpha > 0$, often 0.01. Its derivative is 1 for $a > 0$ and α for $a < 0$, so no unit can die [21, Sec. 6.3.1], [22].

Example 10.2 (A unit that would be dead on half the cars). Let one hidden unit read standardized horsepower in the Auto data [20], with weight $w = -1$ and bias 0, so its pre-activation is $a = -z$. Count the cars that give the unit a zero gradient under ReLU, and the gradient each car passes under a leaky ReLU with $\alpha = 0.01$:

10.3 Python

```

import numpy as np
import pandas as pd

hp_lr = pd.read_csv("data/auto.csv")["horsepower"].dropna()
a_lr = -((hp_lr - hp_lr.mean()) / hp_lr.std()).to_numpy() # pre-activation: weight -1, bias 0

leak_lr = 0.01
d_relu_lr = (a_lr > 0).astype(float)
d_leaky_lr = np.where(a_lr > 0, 1.0, leak_lr)
print(len(a_lr), (d_relu_lr == 0).sum(), np.unique(d_leaky_lr))

392 148 [0.01 1. ]

```

10.4 R

```
hp_lr <- na.omit(read.csv("data/auto.csv")$horsepower)
a_lr <- -as.vector(scale(hp_lr)) # pre-activation: weight -1, bias 0

leak_lr <- 0.01
d_relu_lr <- as.numeric(a_lr > 0)
d_leaky_lr <- ifelse(a_lr > 0, 1, leak_lr)
c(cars = length(a_lr), relu_zero = sum(d_relu_lr == 0), leaky_values = length(unique(d_leaky_lr)))
```

cars	relu_zero	leaky_values
392	148	2

For 392 cars, 148 have $a \leq 0$. Under ReLU the unit passes no gradient for any of them, so training on those cars cannot move its weight. Under the leaky ReLU each of them still passes a gradient of 0.01, which is small but not zero, so the weight can recover.

11 Residual connections

ReLU and He initialization keep each factor of the chain near one. The other way out is to give the gradient a route that avoids the chain.

Definition 11.1 (Residual connection (skip connection, shortcut connection)). A **residual connection** around hidden layer k adds the layer's input to its output, so that the layer computes

$$\tilde{h}^{(k)} = \tilde{h}^{(k-1)} + f(\tilde{a}^{(k)}) \quad \text{instead of} \quad \tilde{h}^{(k)} = f(\tilde{a}^{(k)}),$$

with $\tilde{a}^{(k)}$ as before. The layer then learns a change to its input, the *residual*, rather than a replacement for it [23], [24]. The sum needs the layer's input and output to have the same width, $L_{k-1} = L_k$.

Other names: **skip connection**, **shortcut connection**.

Example 11.1 (Twenty layers with and without the skip). Stack 20 hidden layers of width 4 with tanh units and small random weights, drawn once in R, and pass one input vector through them. Compare the layers $\tilde{h}^{(k)} = f(\tilde{a}^{(k)})$ with the residual layers $\tilde{h}^{(k)} = \tilde{h}^{(k-1)} + f(\tilde{a}^{(k)})$, using the same weights for both:

11.1 R

```
set.seed(3)
width_rc <- 4
# Small weights (sd 0.2), so each layer's change to its input is small.
Ws_rc <- lapply(1:20, function(k) matrix(rnorm(width_rc^2, sd = 0.2), width_rc, width_rc))
x_rc <- c(1, -0.5, 0.25, 2)

h_plain_rc <- h_resid_rc <- x_rc
for (W in Ws_rc) {
  h_plain_rc <- tanh(W %*% h_plain_rc) # replaces the input
  h_resid_rc <- h_resid_rc + tanh(W %*% h_resid_rc) # adds a change to the input
}
# Hand the weights and the input to the Python chunk below, which reads them as ordinary variables.
py$Ws_from_r_rc <- Ws_rc
py$x_from_r_rc <- x_rc
c(input = sqrt(sum(x_rc^2)), plain = sqrt(sum(h_plain_rc^2)), residual = sqrt(sum(h_resid_rc^2)))
```

input	plain	residual
2.304886e+00	1.639422e-08	2.443157e+00

11.2 Python

```
import numpy as np

# The weights and the input come from the R chunk of this example,
# so that both languages use the same numbers.
Ws_py_rc = [np.asarray(W) for W in Ws_from_r_rc]
x_py_rc = np.asarray(x_from_r_rc)

h_plain_py_rc = h_resid_py_rc = x_py_rc
for W in Ws_py_rc:
    h_plain_py_rc = np.tanh(W @ h_plain_py_rc) # replaces the input
    h_resid_py_rc = h_resid_py_rc + np.tanh(W @ h_resid_py_rc) # adds a change to the input
print(np.linalg.norm(x_py_rc), np.linalg.norm(h_plain_py_rc), np.linalg.norm(h_resid_py_rc))
```

2.3048861143232218 1.6394217748097892e-08 2.443157472227823

The input has norm 2.30. After 20 plain layers it has norm 1.6e-08: each layer shrank the signal, and little of the input survives. The residual stack still has norm 2.44, because every layer starts from its input and only adds a correction. The same holds for gradients going backward, since the identity path contributes a factor of 1 at each layer.

Exercise 11.1 (Give the gradient a path around the chain). Exercise 8.1 found that $\tilde{\delta}^{(1)}$ carries one factor of f' for every hidden layer, and Exercise 8.7 made each factor larger. A residual connection (Definition 11.1) attacks the count instead.

Put a residual connection around every hidden layer from $k = 2$ to K , with all hidden widths equal to L , and write $\epsilon^{(k)} = \partial\mathcal{L}/\partial h^{(k)}$ for the gradient with respect to a layer's output.

Differentiate $\tilde{h}^{(k)} = \tilde{h}^{(k-1)} + f(\tilde{a}^{(k)})$ to get the recursion that takes $\epsilon^{(k)}$ to $\epsilon^{(k-1)}$, by way of $\tilde{\delta}^{(k)} = \partial\mathcal{L}/\partial \tilde{a}^{(k)}$. Compare it with the recursion for the same network without the residual connection.

Solution

Solution 11.1. Write $W_{1:L,:}^{(k)}$ for $W^{(k)}$ without its bias row, as in Solution 8.1, and $F^{(k)} = \text{diag}(f'(a^{(k)}))$. The output of the block depends on its input twice: directly, and through $\tilde{a}^{(k)}$. So

$$\frac{\partial h^{(k)}}{\partial \tilde{h}^{(k-1)}} = I + F^{(k)} (W_{1:L,:}^{(k)})^\top,$$

and the chain rule gives the recursion

$$\tilde{\delta}^{(k)} = f'(\tilde{a}^{(k)}) \odot \epsilon^{(k)}, \quad \epsilon^{(k-1)} = \epsilon^{(k)} + W_{1:L,:}^{(k)} \tilde{\delta}^{(k)}.$$

Without the residual connection the second equation loses its first term: $\epsilon^{(k-1)} = W_{1:L,:}^{(k)} \tilde{\delta}^{(k)}$. The residual connection hands the layer below a copy of the gradient it received, and adds the usual matrix-and- f' term on top.

Exercise 11.2 (Give the gradient a path around the chain (part 2)). Use the setup of Exercise 11.1. Unroll your recursion from $\epsilon^{(K)}$ down to $\epsilon^{(1)}$ as a product of $K - 1$ matrix factors. Expand the product, and find the term that carries no factor of f' at all. How many factors of f' does $\tilde{\delta}^{(1)}$ then carry along that term, and how many did it carry in Exercise 8.1?

Solution

Solution 11.2. Substituting the first equation into the second, $\epsilon^{(k-1)} = (I + W_{1:L,:}^{(k)} F^{(k)}) \epsilon^{(k)}$, so

$$\epsilon^{(1)} = (I + W_{1:L,:}^{(2)} F^{(2)}) (I + W_{1:L,:}^{(3)} F^{(3)}) \dots (I + W_{1:L,:}^{(K)} F^{(K)}) \epsilon^{(K)}.$$

Expanding the product gives 2^{K-1} terms, one for each choice of which factors contribute their WF part and which contribute I . Choosing I every time gives $\epsilon^{(K)}$ itself, with no factor of f' and no weight matrix: the gradient arriving at the top of the stack reaches layer 1 at full size. The other terms add to it; the shortest of them passes through one block.

Then $\delta^{(1)} = F^{(1)} \epsilon^{(1)}$ carries **one** factor of f' along that term, where Exercise 8.1's $\delta^{(1)}$ carried K . Along that term the bound 4^{-K} becomes 4^{-1} , whatever the depth.

Exercise 11.3 (Give the gradient a path around the chain (part 3)). Use the setup of Exercise 11.1. With $f = \text{ReLU}$, set every weight in $W^{(2)}, \dots, W^{(K)}$ to zero. What does the network compute, and what is $\epsilon^{(1)}$? Now do the same with $f = \text{explit}$, and say what each block adds to its input^a instead.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Solution

Solution 11.3. With $f = \text{ReLU}$ and $W^{(k)} = 0$, each block has $\tilde{a}^{(k)} = \vec{0}$ and $f(\vec{0}) = \vec{0}$, so $\tilde{h}^{(k)} = \tilde{h}^{(k-1)}$: every block is the identity, and the network computes exactly what its first hidden layer and output layer compute on their own. The recursion of Solution 11.1 gives $\epsilon^{(k-1)} = \epsilon^{(k)}$ at every block, so $\epsilon^{(1)} = \epsilon^{(K)}$. A residual network can start as a shallow one and use its extra layers only where they help. A plain stack has no such default: to pass its input^a through unchanged, each layer would have to learn the identity from its weights.

With $f = \text{explit}$ the same weights give $\text{explit}(0) = 1/2$, so each block adds $1/2$ to every unit, and the $K - 1 = 9$ blocks of the measurement add 4.5 to each entry of $\tilde{h}^{(1)}$ between them.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Exercise 11.4 (Give the gradient a path around the chain (part 4)). Use the setup of Exercise 11.1. Repeating Section 8 on the same 200 sigmoid^a networks, with a residual connection around each of hidden layers 2 to 10, moves the gradient norm^b at layer 1 from 1.11×10^{-6} to 0.95 . The geometric-mean ratio between consecutive hidden layers, leaving out the output layer, falls from 4.22 to 1.10 , and the gradient norm at the output layer rises from 2.12 to 21.1 . Say which of these numbers Exercise 11.2 predicted, and why the hidden layers' gradients are still not all equal. Then use Exercise 11.3 to explain the rise at the output layer, and say what a transformer pairs^c with each residual connection to stop it.

^a<https://morrison-lab.github.io/mds/functions.html#def-logistic-function>

^b<https://morrison-lab.github.io/mds/linear-algebra-inner-products.html#def-norm>

^c<https://morrison-lab.github.io/mds/sets.html#def-ordered-pair>

Solution

Solution 11.4. The measurement uses the same draws as Section 8, so the two runs see the same 200 networks, inputs^a and targets. Without the residual connections it reproduces the norms^b printed there, which the page checks before quoting either set.

```
# The measurement of the vanishing-gradient experiment, on the same draws, with an
# optional residual connection around hidden layers 2 to K. Besides the
# per-layer gradient norms it returns the average norm of  $[1; h^{(K)}]$ , the
# input to the output layer.
```

```
def measure_residual(act, dact, residual, K=10, L=20, D=20, C=3, N=200,
                    seed=0):
    rng = np.random.default_rng(seed)
    widths = [D] + [L] * K + [C]
    norms = np.zeros(K + 1)
    stream = 0.0
    for _ in range(N):
        W = [rng.normal(0, 1 / np.sqrt(widths[k] + 1),
                        (widths[k] + 1, widths[k + 1]))
              for k in range(K + 1)]
        post = [np.concatenate(([1.0], rng.normal(0, 1, D)))]
        pre, outs = [], []
        for k in range(K + 1):
            a = W[k].T @ post[k]
            pre.append(a)
            if k < K:
                out = act(a)
                outs.append(out)
                h = out + post[k][1:] if residual and k >= 1 else out
                post.append(np.concatenate(([1.0], h)))
            stream += np.linalg.norm(post[k])
        e = np.exp(pre[K] - pre[K].max())
        t = np.zeros(C)
        t[rng.integers(C)] = 1.0
        d = e / e.sum() - t
        norms[K] += np.linalg.norm(np.outer(post[K], d))
        eps = W[K][1:, :] @ d
        for k in range(K - 1, -1, -1):
            d = eps * dact(outs[k])
            norms[k] += np.linalg.norm(np.outer(post[k], d))
            if k > 0:
                eps = (eps if residual else 0) + W[k][1:, :] @ d
    return norms / N, stream / N
```

```
sigmoid = lambda a: 1 / (1 + np.exp(-a))
dsigmoid = lambda h: h * (1 - h)
plain_norms, plain_stream = measure_residual(sigmoid, dsigmoid, False)
res_norms, res_stream = measure_residual(sigmoid, dsigmoid, True)
```

Solution 11.2 predicted the first number. The gradient at layer 1 grows by a factor of 8.6×10^5 , from 1.11×10^{-6} to 0.95, and the ratio between consecutive hidden layers falls from 4.22, the 1/4 of Exercise 8.1 made visible, to 1.10. These ratios leave out the output layer, so the first differs a little from the 4.25 of Section 8; with the residual connections the output layer's gradient rises for a separate reason, below.

The hidden layers' gradients still differ, for two reasons. Each $\partial \mathcal{L} / \partial W^{(k)} = [1; \tilde{h}^{(k-1)}] (\tilde{\delta}^{(k)})^\top$ still has its own factor of f' in $\tilde{\delta}^{(k)}$, which Solution 11.2 cannot remove because it sits inside the block, not on the path around it. And its other factor, $\tilde{h}^{(k-1)}$, gets larger towards the output. Layer 1 is the exception: its input^c is x itself, whose entries are not squashed into $(0, 1)$.

That growth is Solution 11.3's sigmoid^d case. Every block adds an output in $(0, 1)$ to the stream, so the input to the output layer has average norm^e 22.8 with the residual connections, against 2.5 without

them. The output layer’s gradient is that input times $\tilde{\delta}^{(K+1)}$, so it rises with it, from 2.12 to 21.1. A deeper stack would push it further. A transformer follows each residual connection with a layer normalization, which rescales each position’s vector to mean^f 0 and variance^g 1 before a learned scale and shift; this is the “Add & Norm” of Attention and Transformers.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^b<https://morrison-lab.github.io/mds/linear-algebra-inner-products.html#def-norm>

^c<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

^d<https://morrison-lab.github.io/mds/functions.html#def-logistic-function>

^e<https://morrison-lab.github.io/mds/linear-algebra-inner-products.html#def-norm>

^f<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

^g<https://morrison-lab.github.io/pds/variance-covariance.html#def-variance>

Exercise 11.5 (Give the gradient a path around the chain (part 5)). Use the setup of Exercise 11.1. The sum in Definition 11.1 needs $L_{k-1} = L_k$. Suggest how to keep a path around a layer that changes the width.

Solution

Solution 11.5. Replace the identity on the path around the layer with a matrix:

$$\tilde{h}^{(k)} = P^{(k)} \tilde{h}^{(k-1)} + f(\tilde{a}^{(k)}),$$

where $P^{(k)}$ is $L_k \times L_{k-1}$, either learned or fixed. He and colleagues describe a learned linear projection of this kind and a fixed one that pads the shorter vector with zeros, keeping the identity wherever the width does not change [23, Secs. 3.2–3.3], and find the learned projection only slightly better [23, Sec. 4.1]. The path around the layer then multiplies the gradient by $(P^{(k)})^\top$ once, rather than by a chain of matrices and f' factors.

i Remark (Shattered gradients)

Remark 11.1 (Shattered gradients). A gradient can be the right size and still be useless. Balduzzi and colleagues compared the gradients of deep rectifier networks at nearby inputs. In a plain network the correlation between them falls like 2^{-L} with depth L , until the gradient resembles white noise. Residual connections like those of Definition 11.1 slow this to $(3/4)^L$, and with batch normalization added the correlation falls only like $1/\sqrt{L}$ [25, Secs. 3.1–3.3]. Descent on such a gradient makes no consistent progress, and checking the gradient’s norm, which catches vanishing and exploding gradients, does not catch this.

12 A whole minibatch at once

Everything so far handled one training example. Gradient descent already argued for minibatches on statistical grounds — a gradient averaged over B examples is a better estimate than a gradient from one.

A second argument decides the batch size in practice. The same formulas, with vectors replaced by matrices, run far faster per example than a loop over the same examples would.

Exercise 12.1 (Do a whole minibatch at once). Gradient descent established why training uses minibatches rather than single examples. This exercise does the bookkeeping: every formula so far handled one example, and a minibatch of B of them should not mean running those formulas B times in a loop.

Stack the examples as **columns**. Write $\mathbf{X} \in \mathbb{R}^{(p+1) \times B}$ for the inputs^a and $T \in \mathbb{R}^{C \times B}$ for the one-hot targets, and take the loss to be the mean over the batch, $\mathcal{L} = \frac{1}{B} \sum_{i=1}^B \mathcal{L}_i$.

Capital letters denote the batched version of the matching lower-case vector: A , H and Y for $\tilde{a}$, $\tilde{h}$ and

y . The output pre-activation b is the one exception — its capital is already the batch size, so write it $Z \in \mathbb{R}^{C \times B}$.

Rewrite the four forward equations of Exercise 2.1 with $\mathbf{X}$ in place of $\tilde{x}$, and give the shape of each result. Say what happens to the bracket $[1; h]$.

^a<https://morrison-lab.github.io/sds/model-validation.html#def-data-set>

Solution

Solution 12.1.

$$\begin{aligned} A &= W^\top \mathbf{X} \in \mathbb{R}^{L \times B}, & H &= f(A) \in \mathbb{R}^{L \times B}, \\ Z &= U^\top \begin{bmatrix} \mathbf{1}^\top \\ H \end{bmatrix} \in \mathbb{R}^{C \times B}, & Y &= g(Z) \in \mathbb{R}^{C \times B}. \end{aligned}$$

The bracket becomes a **row of ones** stacked on top of H , giving an $(L + 1) \times B$ matrix: each column still carries its own constant 1. f is elementwise as before; g is softmax applied **down each column separately**, since each column is one example's own distribution over the C classes.

Exercise 12.2 (Do a whole minibatch at once (part 2)). Use the setup of Exercise 12.1. $\Delta^b = \partial \mathcal{L} / \partial Z$ collects the output errors of the whole batch. Give it, including whatever the $\frac{1}{B}$ contributes.

Solution

Solution 12.2. Differentiating the mean^a rather than the sum divides everything by B :

$$\Delta^b = \frac{1}{B}(Y - T) \in \mathbb{R}^{C \times B}.$$

Column i is that example's $\tilde{\delta}^b$, scaled by $1/B$. Carrying the $1/B$ here rather than at the end is the usual choice, because it means^b every gradient downstream comes out already averaged.

^a<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

^b<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

Exercise 12.3 (Do a whole minibatch at once (part 3)). Use the setup of Exercise 12.1. Give $\partial \mathcal{L} / \partial U$ and $\partial \mathcal{L} / \partial W$ as matrix products^a. Confirm that they are the same shape as U and W — **not** B times larger.

^a<https://morrison-lab.github.io/mds/linear-algebra-matrices.html#def-matrix-mult>

Solution

Solution 12.3.

$$\frac{\partial \mathcal{L}}{\partial U} = \begin{bmatrix} \mathbf{1}^\top \\ H \end{bmatrix} (\Delta^b)^\top \in \mathbb{R}^{(L+1) \times C}, \quad \frac{\partial \mathcal{L}}{\partial W} = \mathbf{X} (\Delta^a)^\top \in \mathbb{R}^{(p+1) \times L},$$

with Δ^a from Solution 12.6. $(L + 1) \times B$ against $B \times C$ gives $(L + 1) \times C$, and $(p + 1) \times B$ against $B \times L$ gives $(p + 1) \times L$: both match their parameter matrix exactly.

They must. There is one gradient per parameter no matter how many examples produced it.

Exercise 12.4 (Do a whole minibatch at once (part 4)). Use the setup of Exercise 12.1. In the single-example version those two gradients were outer products^a. Say what happened to the sum over the batch, and where the batched expression^b now performs it.

^a<https://morrison-lab.github.io/mds/linear-algebra-inner-products.html#def-outer-product>

^b<https://morrison-lab.github.io/mds/expressions-algorithms.html#def-expression>

Solution

Solution 12.4. In Exercise 12.4, the sum over the batch became the inner dimension of the matrix product^a.

$[\mathbf{1}^\top; H](\Delta^b)^\top$ computes $\frac{1}{B} \sum_{i=1}^B [1; h_i](\delta_i^b)^\top$, because a matrix product sums over the shared index — and here the shared index is the example. The $\frac{1}{B}$ is the one Solution 12.2 included in Δ^b , so what comes out is the average over the batch rather than the sum. The loop is still there. We have given it to the routine that is best at it. Every optimization anyone has made to dense matrix multiplication, over several decades, now applies to averaging gradients over a batch.

^a<https://morrison-lab.github.io/mds/linear-algebra-matrices.html#def-matrix-mult>

Exercise 12.5 (Do a whole minibatch at once (part 5)). Use the setup of Exercise 12.1.

Exercise 3.4 found that one example gives a rank-one gradient for U . Give the largest possible rank^a of $\partial\mathcal{L}/\partial U$ for a batch of B , and say what batch size is needed before that limit stops applying.

^a<https://morrison-lab.github.io/mds/linear-algebra-subspaces.html#def-rank>

Solution

Solution 12.5. A product of an $(L+1) \times B$ matrix with a $B \times C$ one has rank^a at most $\min(L+1, C, B)$. With $C = 10$ outputs, the batch stops being the limiting factor once $B \geq 10$. Below that, the batch itself limits how many directions in U a single step can move, regardless of how informative the examples are — which is a second reason, alongside the arithmetic of Exercise 12.7, that batches of one are a poor default.

^a<https://morrison-lab.github.io/mds/linear-algebra-subspaces.html#def-rank>

Exercise 12.6 (Do a whole minibatch at once (part 6)). Use the setup of Exercise 12.1.

Write the whole backward pass out, and check that every line is a matrix product^a or an elementwise product, with no explicit loop over examples anywhere in it.

^a<https://morrison-lab.github.io/mds/linear-algebra-matrices.html#def-matrix-mult>

Solution

Solution 12.6.

$$\Delta^b = \frac{1}{B}(Y - T), \quad \frac{\partial \mathcal{L}}{\partial U} = \begin{bmatrix} \mathbf{1}^\top \\ H \end{bmatrix} (\Delta^b)^\top,$$

$$\Delta^h = U_{1:L,:} \Delta^b, \quad \Delta^a = \Delta^h \odot f'(A), \quad \frac{\partial \mathcal{L}}{\partial W} = \mathbf{X}(\Delta^a)^\top.$$

Three matrix products^a and one elementwise product. This is the whole of back-propagation for a one-hidden-layer network, and it is the form an implementation actually takes: no loop over examples appears anywhere, and the only thing that changed from the single-example version is that vectors became matrices with B columns.

^a<https://morrison-lab.github.io/mds/linear-algebra-matrices.html#def-matrix-mult>

Exercise 12.7 (Work out why a batch is faster per example). Exercise 12.1 shows that a batch changes no formula and no gradient — B examples produce the average of the B single-example gradients, as a loop would have. So the batch gives no mathematical benefit. It gives an arithmetic benefit, and this

exercise says what.

Keep $p = 784$, $L = 100$, $C = 10$, so $P = 79,510$, and suppose the parameters are stored in 32-bit floats. Give the number of multiply-accumulates in the forward pass for a batch of B , and the number of bytes of **weights** that have to be read to perform it. Say which of the two depends on B .

Solution

Solution 12.7. Every weight is used once per example, so the forward pass is $PB = 79,510B$ multiply-accumulates.

The weights are read **once** regardless of B : $4P = 318,040$ bytes. The arithmetic scales with the batch; the memory traffic for the weights does not.

Exercise 12.8 (Work out why a batch is faster per example (part 2)). Use the setup of Exercise 12.7. Form the ratio: how many floating-point operations does the hardware perform per byte of weights it reads? (Count a multiply-accumulate as two operations.) Give the formula and evaluate it at $B = 1, 8, 32, 128$.

Solution

Solution 12.8. Two operations per multiply-accumulate gives

$$\frac{2PB}{4P} = \frac{B}{2} \text{ operations per byte,}$$

which is independent^a of the network's size entirely — the P cancels.

B	operations per byte of weights
1	0.5
8	4
32	16
128	64

^a<https://morrison-lab.github.io/pds/independence.html#def-indpt>

Exercise 12.9 (Work out why a batch is faster per example (part 3)). Use the setup of Exercise 12.7. An accelerator has some fixed ratio of arithmetic throughput to memory bandwidth, in operations per byte. Explain, in terms of your Exercise 12.8 answer, why small batches leave it idle and what changes as B grows.

Solution

Solution 12.9. With $B = 1$ the hardware performs half an operation for every byte it fetches, so it spends nearly all its time waiting on memory: the arithmetic units are idle, and the step takes as long as reading the weights takes.

As B grows, the same fetched weights serve more examples, and the ratio rises linearly. Once $B/2$ exceeds the machine's own operations-per-byte ratio, the weights arrive faster than the arithmetic consumes them and the step is bounded by compute — which is the regime the hardware was designed for. Below that point, increasing B costs **almost no extra wall-clock time**, because the extra arithmetic fills gaps that were previously idle.

Solution 12.7 counted only the weights, though, and Solution 12.10 is about to show that the activations outweigh them. The activations scale with B , so they cancel out of the ratio the way the weights do not, and the true intensity saturates rather than rising without limit: with 895 floats of activation per

example, it is

$$\frac{2PB}{4P + 4 \cdot 895B} \xrightarrow{B \rightarrow \infty} \frac{2P}{4 \cdot 895} \approx 44 \text{ operations per byte,}$$

reaching 26 at $B = 128$ rather than the 64 the table gives. A network of this shape therefore cannot be driven compute-bound on an accelerator whose own ratio exceeds about 44, however large the batch. The direction of Solution 12.8’s argument survives — raising B does raise the intensity, steeply at first — but the upper limit is set by the shape of the network, not by the batch size.

This is the real content of “minibatching is faster”: a batch is more arithmetic per step and less wall-clock time per example, because a matrix-matrix product reuses data that a matrix-vector product fetches and throws away.

Exercise 12.10 (Work out why a batch is faster per example (part 4)). Use the setup of Exercise 12.7. Exercise 2.3 counted the stored activations. At $B = 128$, compare the bytes of activations with the bytes of weights, and say which one bounds the batch size in practice.

Solution

Solution 12.10. At $B = 128$ the stored activations are $128 \times (784 + 100 + 10 + 1) \times 4 = 458,240$ bytes, against 318,040 bytes of weights.

The activations are already larger, and unlike the weights they grow with B — and, in a deep network, with the number of layers as well, since every layer’s activations are held until the backward pass reaches them. So activation memory, not the model, sets the largest batch that fits, and a practitioner meets that limit first.

Exercise 12.11 (Work out why a batch is faster per example (part 5)). Use the setup of Exercise 12.7. Against these benefits, a larger batch means^a fewer updates for the same amount of arithmetic. The minibatch gradient is an average of B independent^b estimates^c, so say how its standard error^d scales with B , and how the work per step scales. What does the comparison say about doubling the batch size repeatedly?

^a<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

^b<https://morrison-lab.github.io/pds/independence.html#def-indpt>

^c<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

^d<https://morrison-lab.github.io/sds/estimation.html#def-SE>

Solution

Solution 12.11. The batch gradient is a mean^a of B independent^b unbiased estimates^c (gradient descent), so its standard error^d falls like $1/\sqrt{B}$ while the work per step rises like B :

B	noise	work per step
1	1	1×
4	0.5	4×
16	0.25	16×
64	0.125	64×

Quadrupling the batch halves the noise and quadruples the cost. Doubling repeatedly therefore has sharply diminishing returns: past some point you are paying linearly for a square-root improvement in an estimate^e that was never the bottleneck — gradient descent tolerates a noisy direction, as gradient descent shows.

^a<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

^b<https://morrison-lab.github.io/pds/independence.html#def-indpt>

^c<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

^d<https://morrison-lab.github.io/sds/estimation.html#def-SE>

^e<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

Exercise 12.12 (Work out why a batch is faster per example (part 6)). Use the setup of Exercise 12.7. Put Exercise 12.9 and Exercise 12.11 together into a statement about how to choose B , and say which part of it is a fact about the estimator^a and which is a fact about the machine.

^a<https://morrison-lab.github.io/sds/estimation.html#def-estimator>

Solution

Solution 12.12. Raise B until the arithmetic intensity is near its upper limit, and no further than the memory allows; past that point extra examples give $\sqrt{\cdot}$ and cost B .

The $1/\sqrt{B}$ is a fact about **sampling**: the variance^a of a mean^b of B independent^c estimates^d is σ^2/B , which is an exact identity at every B and needs only that the variance is finite. It would hold on any machine ever built. The operations-per-byte ratio and the activation-memory limit are facts about the machine, and they change with every hardware generation. So batch size is the hyperparameter that moves when the hardware changes — and it is a useful habit to ask of any training recipe which of its numbers are facts about the problem and which are facts about the machine. The two are not independent: the common rules for changing B scale the learning rate with it, linearly or as $\sqrt{B}$, precisely because a less noisy gradient can support a longer step.

^a<https://morrison-lab.github.io/pds/variance-covariance.html#def-variance>

^b<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

^c<https://morrison-lab.github.io/pds/independence.html#def-indpt>

^d<https://morrison-lab.github.io/sds/estimation.html#def-estimate>

Exercise 12.13 (Fit a big batch in a small memory). Solution 12.7 found that the activations of a batch, which grow with the batch size B , are what limit B in practice. Suppose the activations of only $m < B$ examples fit in memory at once, yet you want the gradient step of a batch of B . Describe a way to take that step using m examples at a time, and say what you add up and when you take the step.

Solution

Solution 12.13. Split the batch into groups of at most m examples. For each group in turn, run forward and backward propagation, and add the group's summed per-example gradients to a running total. Then free that group's activations and move to the next group. Take one gradient step only after the last group, with the total divided by B . This is **gradient accumulation**. Memory holds the activations of m examples, plus one extra array the size of the parameters for the total.

Exercise 12.14 (Fit a big batch in a small memory (part 2)). Use the setup of Exercise 12.13. Show that your procedure gives exactly the gradient of the full batch.

Solution

Solution 12.14. The batch gradient is an average of per-example gradients, which is a sum divided by B :

$$\nabla J(\vec{\beta}) = \frac{1}{B} \sum_{i=1}^B \nabla \ell_i(\vec{\beta}).$$

A sum can be split into groups, $\mathcal{G}_1, \dots, \mathcal{G}_K$, and added in any order:

$$\sum_{i=1}^B \nabla \ell_i(\vec{\beta}) = \sum_{k=1}^K \sum_{i \in \mathcal{G}_k} \nabla \ell_i(\vec{\beta}).$$

Each group's inner sum needs only that group's activations, and the parameters do not change until the step, so the accumulated total equals the full-batch sum exactly, up to floating-point rounding. It is the same step a memory large enough for B examples would take.

Exercise 12.15 (Fit a big batch in a small memory (part 3)). Use the setup of Exercise 12.13. Suppose B is not a multiple of m , so the last group is smaller. What goes wrong if you average the group means^a instead of what you added up in Exercise 12.13?

^a<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

Solution

Solution 12.15. Each group's mean^a divides by its own size n_k . Averaging the K group means^b gives every group the weight $1/K$, so an example in the small last group counts for more than one in a full group. The result is not the batch gradient. Dividing the accumulated sum by B , as in Solution 12.13, weights every example by $1/B$.

^a<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

^b<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

Exercise 12.16 (Fit a big batch in a small memory (part 4)). Use the setup of Exercise 12.13. Check Exercise 12.14 and Exercise 12.15 for the sample mean squared error (sMSE) of a linear model on $B = 100$ simulated examples with $m = 32$.

Solution

Solution 12.16.

12.1 R

```

# Simulate B = 100 examples of a linear model with 3 coefficients.
set.seed(571)
B <- 100
m <- 32
X <- matrix(rnorm(B * 3), nrow = B)
y <- X %*% c(1, -2, 0.5) + rnorm(B)
coef <- c(0, 0, 0)

# The summed gradient of the squared error over some rows, at the current
# coefficients: the sum over examples of 2 x (x'coef - y).
grad_sum <- function(rows) {
  2 * crossprod(X[rows, , drop = FALSE], X[rows, , drop = FALSE] %*% coef - y[rows])
}

# The full-batch gradient, all rows at once.
full <- grad_sum(1:B) / B

# Accumulate over groups of at most m rows, then divide by B.
total <- numeric(3)
means <- list()
starts <- seq(1, B, by = m)
for (s in starts) {
  rows <- s:min(s + m - 1, B)
  total <- total + grad_sum(rows)
  means[[length(means) + 1]] <- grad_sum(rows) / length(rows)
}
accumulated <- total / B
last_n <- B - m * (length(starts) - 1)

# Averaging the group means gives every group the same weight.
mean_of_means <- Reduce(`+`, means) / length(means)

isTRUE(all.equal(c(full), accumulated))

[1] FALSE

isTRUE(all.equal(c(full), c(mean_of_means)))

[1] FALSE

```

12.2 Python

```

import numpy as np

# Simulate B = 100 examples of a linear model with 3 coefficients.
rng = np.random.default_rng(571)
B, m = 100, 32
X = rng.normal(0, 1, size=(B, 3))
y = X @ np.array([1, -2, 0.5]) + rng.normal(0, 1, size=B)
coef = np.zeros(3)

# The summed gradient of the squared error over some rows, at the current
# coefficients: the sum over examples of 2 x (x'coef - y).
def grad_sum(rows):
    return 2 * X[rows].T @ (X[rows] @ coef - y[rows])

# The full-batch gradient, all rows at once.
full = grad_sum(slice(0, B)) / B

# Accumulate over groups of at most m rows, then divide by B.
total = np.zeros(3)
means = []
for s in range(0, B, m):
    rows = slice(s, min(s + m, B))
    total += grad_sum(rows)
    means.append(grad_sum(rows) / len(y[rows]))
accumulated = total / B

# Averaging the group means gives every group the same weight.
mean_of_means = np.mean(means, axis=0)

print(np.allclose(full, accumulated), np.allclose(full, mean_of_means))

```

True False

The accumulated gradient matches the full-batch gradient, and the average of the group means^a does not. Here the batch splits into 3 full groups of 32 and a last group of 4, so with the group means averaged, each example in the last group counts 6.25 times as heavily as in the batch.

A caveat: this equivalence needs the loss of each example to depend only on that example. A layer that computes statistics across the batch, such as batch normalization, couples the examples, so accumulating small groups does not reproduce the full-batch step.

^a<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean>

i Watch

- The CS231N lecture shows vectorized backpropagation, from 1:05:07^a [11], Lecture 4.
- Starmer explains the matrix algebra of a network^b [26].

^a<https://www.youtube.com/watch?v=25zD5qJHYsk&t=3907s>

^b<https://www.youtube.com/watch?v=ZTt9gsGcdDo>

i Watch

Hutchinson covers minibatched backprop^a (11 min) [3].

^ahttps://facultyweb.cs.wvu.edu/~hutchib2/video_lectures/data371/#mb_backprop

13 Gradients with respect to the input

Back-propagation differentiates the loss with respect to the *weights*, but the chain rule works for any variable it is applied to. Run the same backward sweep one step further, across W , and it returns the gradient with respect to the **input**:

$$\frac{\partial \mathcal{L}}{\partial x_{1:p}} = W_{1:p, :} \delta^a.$$

This is the crossing Exercise 4.1 made across U , with the row for the constant dropped for the same reason: the constant is not an input anyone can change.

That derivative answers a different question — not “how should the model change?” but “which parts of *this input* is the prediction sensitive to?” Starting the sweep from the indicator of one class in place of $y - \tilde{t}$ gives the gradient of that class’s score instead of the loss’s.

Definition 13.1 (Saliency map). For an image classifier, a **saliency map** is the gradient of the predicted class’s score with respect to each pixel of one input image, displayed as a picture. A large magnitude at a pixel means a small change to that pixel would move the prediction noticeably.

Example 13.1 (Which pixels decide a 3 versus an 8). The `digits` data of scikit-learn [27] are 8×8 grey-scale images of handwritten digits. Fit a logistic regression that tells 3s from 8s, and take one 8. The gradient of its predicted probability of “8” with respect to the 64 pixels is the saliency map of Definition 13.1. For a logistic regression, $p = \text{expit}(\tilde{w} \cdot \tilde{x} + b)$, so the gradient is $p(1 - p) \tilde{w}$:

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_digits
from sklearn.linear_model import LogisticRegression

digits_sm = load_digits()
keep_sm = np.isin(digits_sm.target, [3, 8])
X_sm = digits_sm.data[keep_sm] / 16.0      # pixel values scaled to [0, 1]
y_sm = (digits_sm.target[keep_sm] == 8).astype(int)

clf_sm = LogisticRegression(max_iter=1000).fit(X_sm, y_sm)

# Take the first image of an 8, and compute p(1 - p) * w, the gradient of p with respect to the pixels
i_sm = int(np.flatnonzero(y_sm == 1)[0])
p_sm = clf_sm.predict_proba(X_sm[i_sm:i_sm + 1])[0, 1]
saliency_sm = p_sm * (1 - p_sm) * clf_sm.coef_[0]

fig, axes = plt.subplots(1, 2, figsize=(6, 3), layout="constrained")
axes[0].imshow(X_sm[i_sm].reshape(8, 8), cmap="Greys");
axes[0].set_title("an 8");
bound_sm = np.abs(saliency_sm).max()
axes[1].imshow(saliency_sm.reshape(8, 8), cmap="coolwarm", vmin=-bound_sm, vmax=bound_sm);
axes[1].set_title("saliency map");
for ax in axes:
    ax.set_xticks([]);
    ax.set_yticks([]);
plt.show()
```

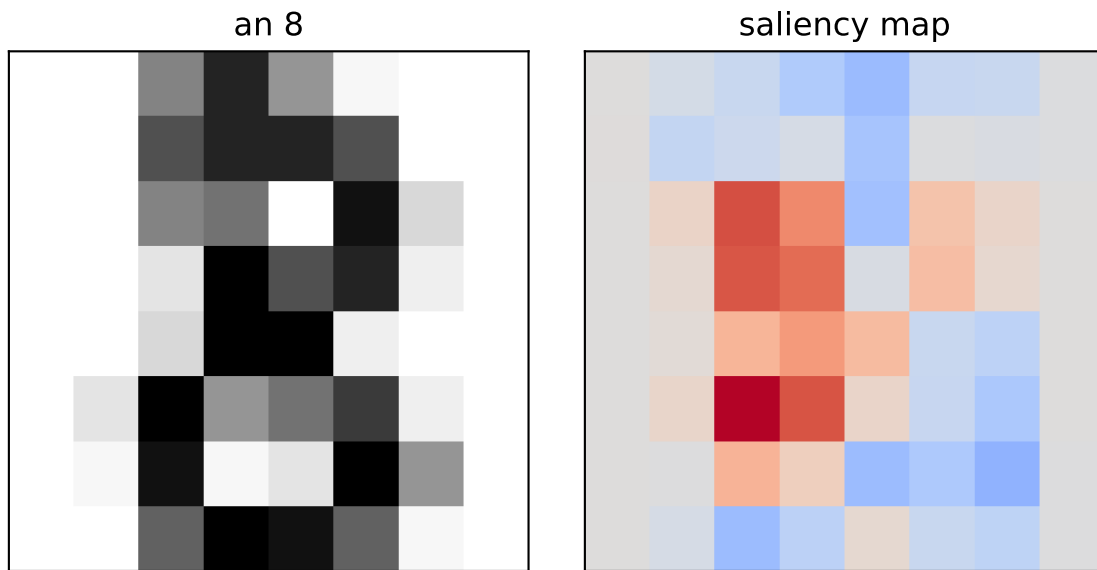


Figure 2: One image of an 8 from the scikit-learn digits data (left) and its saliency map for the class 8 (right). Red pixels would raise the predicted probability of 8 if they were darker; blue pixels would lower it.

The model gives this image a probability of 0.985 of being an 8. Because the model is linear in the pixels, the map is the weight vector times the constant $p(1 - p)$, so its pattern is the same for every image and only its size changes with the confidence: it is largest for images the model is unsure about, and nearly zero for images it is sure of. For a deep network, where the gradient depends on the image, the map differs from image to image.

i Read

GeeksforGeeks explains saliency maps^a [28].

^a<https://www.geeksforgeeks.org/machine-learning/what-is-saliency-map/>

A saliency map costs one backward sweep and one extra matrix-vector product, because the computation already exists for training.

14 Checking the gradient in code

The code below implements the formulas of this chapter and then checks them. The code nudges each weight by $\pm\epsilon$ and re-evaluates the loss, which gives a central finite-difference estimate of that weight's partial derivative. The analytic gradient should agree with it to several decimal places. The check earns its place because a wrong gradient is not a loud failure: as long as it points roughly downhill it still *descends*, so the loss falls, the run looks healthy, and the only symptom is that it converges slower or to a worse optimum than it should. Nothing in the training curve distinguishes that from a hard problem.

The problem is XOR, which Neural Networks uses to motivate the hidden layer: four points, not linearly separable, and so unlearnable by anything without one.

14.0.1 Python

```
import numpy as np

def forward(W, U, X):
```

```

a = X @ W                                # a = W'x, one row per example
h = 1 / (1 + np.exp(-a))                 # h = f(a), logistic
hb = np.column_stack([np.ones(len(h)), h])
b = hb @ U                                # b = U'[1; h]
b = b - b.max(axis=1, keepdims=True)
p = np.exp(b)
return h, hb, p / p.sum(axis=1, keepdims=True)    # y = g(b), softmax

def loss(W, U, X, t):
    _, y = forward(W, U, X)
    return -np.mean(np.log(y[np.arange(len(t)), t]))

def gradients(W, U, X, t):
    h, hb, y = forward(W, U, X)
    d_b = y.copy()
    d_b[np.arange(len(t)), t] -= 1          # softmax + cross-entropy
    d_b /= len(t)
    d_h = d_b @ U[1:].T                    # drop the bias row
    d_a = d_h * h * (1 - h)                # f'(a) = h(1-h)
    return X.T @ d_a, hb.T @ d_b

rng = np.random.default_rng(0)
X = np.array([[1., 0, 0], [1., 0, 1], [1., 1, 0], [1., 1, 1]]) # bias column
t = np.array([0, 1, 1, 0]) # XOR
W = rng.normal(scale=1.0, size=(3, 4))
U = rng.normal(scale=1.0, size=(5, 2))

# Does the derivation match the function it claims to differentiate?
gW, gU = gradients(W, U, X, t)
eps, worst = 1e-6, 0.0
for M, G in ((W, gW), (U, gU)):
    for i in range(M.shape[0]):
        for j in range(M.shape[1]):
            keep = M[i, j]
            M[i, j] = keep + eps; hi = loss(W, U, X, t)
            M[i, j] = keep - eps; lo = loss(W, U, X, t)
            M[i, j] = keep
            worst = max(worst, abs((hi - lo) / (2 * eps) - G[i, j]))
print("largest gap, analytic vs finite difference:", f"{worst:.2e}")

largest gap, analytic vs finite difference: 1.57e-10

for step in range(20_001):
    gW, gU = gradients(W, U, X, t)
    W -= 0.5 * gW
    U -= 0.5 * gU
    if step % 5_000 == 0:
        print(step, round(loss(W, U, X, t), 4))

0 1.4194
5000 0.0019
10000 0.0009
15000 0.0006
20000 0.0004

print("predictions:", forward(W, U, X)[2].argmax(axis=1), "targets:", t)

predictions: [0 1 1 0] targets: [0 1 1 0]

```

14.0.2 R

```
# nolint start: object_name_linter. W, U, X and M match the prose.
forward <- function(W, U, X) {
  a <- X %*% W                                # a = W'x, one row per example
  h <- 1 / (1 + exp(-a))                      # h = f(a), logistic
  hb <- cbind(1, h)
  b <- hb %*% U                                # b = U'[1; h]
  p <- b - apply(b, 1, max)
  p <- exp(p)
  list(h = h, hb = hb, y = p / rowSums(p))    # y = g(b), softmax
}

loss <- function(W, U, X, t) {
  y <- forward(W, U, X)$y
  -mean(log(y[cbind(seq_along(t), t)]))
}

gradients <- function(W, U, X, t) {
  f <- forward(W, U, X)
  d_b <- f$y
  d_b[cbind(seq_along(t), t)] <- d_b[cbind(seq_along(t), t)] - 1
  d_b <- d_b / length(t)
  d_h <- d_b %*% t(U[-1, , drop = FALSE])    # drop the bias row
  d_a <- d_h * f$h * (1 - f$h)               # f'(a) = h(1-h)
  list(W = t(X) %*% d_a, U = t(f$hb) %*% d_b)
}

set.seed(0)
X <- cbind(1, c(0, 0, 1, 1), c(0, 1, 0, 1))  # bias column
t <- c(1L, 2L, 2L, 1L)                      # XOR, as 1-based class labels
W <- matrix(rnorm(12), 3, 4)
U <- matrix(rnorm(10), 5, 2)

# Does the derivation match the function it claims to differentiate?
g <- gradients(W, U, X, t)
eps <- 1e-6
worst <- 0
for (nm in c("W", "U")) {
  M <- get(nm)
  for (i in seq_len(nrow(M))) for (j in seq_len(ncol(M))) {
    keep <- M[i, j]
    M[i, j] <- keep + eps
    assign(nm, M)
    hi <- loss(W, U, X, t)
    M[i, j] <- keep - eps
    assign(nm, M)
    lo <- loss(W, U, X, t)
    M[i, j] <- keep
    assign(nm, M)
    worst <- max(worst, abs((hi - lo) / (2 * eps) - g[[nm]][i, j]))
  }
}
cat("largest gap, analytic vs finite difference:",
    format(worst, digits = 3), "\n")
```

largest gap, analytic vs finite difference: 6.44e-11

```

for (step in 0:20000) {
  g <- gradients(W, U, X, t)
  W <- W - 0.5 * g$W
  U <- U - 0.5 * g$U
  if (step %% 5000 == 0) cat(step, round(loss(W, U, X, t), 4), "\n")
}

```

```

0 0.6982
5000 0.0019
10000 9e-04
15000 6e-04
20000 4e-04

```

```

cat("predictions:", max.col(forward(W, U, X)$y), " targets:", t, "\n")

```

```

predictions: 1 2 2 1 targets: 1 2 2 1

```

```

# nolint end

```

14.0.3 Julia

```

using Random, Statistics

```

```

function forward(W, U, X)
  a = X * W                                # a = W'x, one row per example
  h = 1 ./ (1 + exp.(-a))                  # h = f(a), logistic
  hb = hcat(ones(size(h, 1)), h)
  b = hb * U                                # b = U'[1; h]
  b = b .- maximum(b, dims = 2)
  p = exp.(b)
  return h, hb, p ./ sum(p, dims = 2)      # y = g(b), softmax
end;

```

```

function loss(W, U, X, t)
  _, _, y = forward(W, U, X)
  -mean(log(y[i, t[i]])) for i in eachindex(t)
end;

```

```

function gradients(W, U, X, t)
  h, hb, y = forward(W, U, X)
  d_b = copy(y)
  for i in eachindex(t)
    d_b[i, t[i]] -= 1                      # softmax + cross-entropy
  end
  d_b ./= length(t)
  d_h = d_b * U[2:end, :]'                # drop the bias row
  d_a = d_h .* h .* (1 - h)               # f'(a) = h(1-h)
  return X' * d_a, hb' * d_b
end;

```

```

function gradient_check(W, U, X, t; eps = 1e-6)
  gW, gU = gradients(W, U, X, t)
  worst = 0.0
  for (M, G) in ((W, gW), (U, gU)), i in axes(M, 1), j in axes(M, 2)
    keep = M[i, j]
    M[i, j] = keep + eps; hi = loss(W, U, X, t)
    M[i, j] = keep - eps; lo = loss(W, U, X, t)

```

```

        M[i, j] = keep
        worst = max(worst, abs((hi - lo) / (2 * eps) - G[i, j]))
    end
    return worst
end;

function train!(W, U, X, t; h = 0.5, steps = 20_000)
    for step in 0:steps
        gW, gU = gradients(W, U, X, t)
        W .-= h .* gW
        U .-= h .* gU
        step % 5_000 == 0 && println(step, " ", round(loss(W, U, X, t), digits = 4))
    end
end;

Random.seed!(0);
# bias column
X = [1.0 0 0; 1.0 0 1; 1.0 1 0; 1.0 1 1];
# XOR, as 1-based class labels
t = [1, 2, 2, 1];
W = randn(3, 4);
U = randn(5, 2);

# Does the derivation match the function it claims to differentiate?
println("largest gap, analytic vs finite difference: ", gradient_check(W, U, X, t))

```

largest gap, analytic vs finite difference: 1.0225707780531224e-10

```
train!(W, U, X, t)
```

```

0 0.6997
5000 0.0021
10000 0.001
15000 0.0006
20000 0.0005

```

```

_, _, y = forward(W, U, X);
println("predictions: ", [argmax(y[i, :]) for i in axes(y, 1)], " targets: ", t)

```

predictions: [1, 2, 2, 1] targets: [1, 2, 2, 1]

In all three, the analytic gradient agrees with the finite difference to about ten decimal places. That is roughly the best a central difference can do, and the reason is worth knowing, because shrinking ε — the obvious way to sharpen the agreement — makes the agreement worse. Two errors pull in opposite directions as ε shrinks. Truncation error — the $O(\varepsilon^2)$ remainder in the Taylor expansion the difference quotient comes from — falls. Cancellation error rises: the two losses agree in their leading digits, so subtracting them throws those digits away, and dividing the remainder by a smaller ε magnifies what is left. Sweeping ε over each tab's check — that block with `eps` replaced and nothing else changed — gives

worst gap	$\varepsilon = 10^{-2}$	10^{-4}	10^{-6}	10^{-8}	10^{-10}	10^{-12}
Python	9.1×10^{-7}	9.2×10^{-11}	1.6×10^{-10}	2.0×10^{-8}	1.2×10^{-6}	1.5×10^{-4}
R	7.8×10^{-7}	7.8×10^{-11}	6.4×10^{-11}	6.9×10^{-9}	7.6×10^{-7}	7.2×10^{-5}
Julia	4.1×10^{-7}	4.2×10^{-11}	1.0×10^{-10}	1.0×10^{-8}	7.6×10^{-7}	9.7×10^{-5}

The digits differ across the three, because each draws its own weights. The shape does not, and the shape is the point: every row is a valley rather than a slope, bottoming out at $\varepsilon = 10^{-4}$ for Python, 10^{-6} for R and 10^{-4} for Julia, and climbing from there, until at 10^{-12} the check is less accurate than it was at 10^{-2} .

A gradient check that comes back disappointing is therefore not evidence of a wrong gradient until we have varied ε . And $\varepsilon = 10^{-6}$, the value the three tabs above use, sits near the floor. Its worst gap is within a factor of 2.5 of the best in every row. That is a small margin, and that is the useful part: at 10^{-6} the two error terms are still close to balanced, so the check is near its best and the penalty for the habitual value is mild. Cancellation runs away from 10^{-8} onward, and that costs orders of magnitude.

The three tabs print different losses at step 0 for one reason only: each language’s generator produces different initial weights from the same scale-1 normal. The size of the gradient gap and the final predictions do reproduce. All three recover XOR exactly, though they spell the answer differently. Python labels its classes from 0, and R and Julia label from 1. The label base is an indexing convention and nothing more: both conventions select the same two softmax columns, so it cannot move the loss.

Two limits of this example are worth naming. Four points is not a data set, so there is nothing here to overfit and no held-out set to check against — success means memorizing the training data, which is the one setting where that is the goal. And full-batch descent over four points is not how we train a network in practice. See [A whole minibatch at once](#) above.

i Versions

Rendered with NumPy 2.5.4 on Python 3.13.18, R 4.6.1, and Julia 1.13.3. The gradient-check magnitude is a property of the arithmetic and should reproduce anywhere; the per-step losses depend on each language’s random number generator and will not match across the three tabs.

15 Where this goes next

This chapter leads to three topics.

Nobody writes these formulas any more. Solution 6.5 names the general procedure: reverse-mode automatic differentiation, which computes the derivative of one scalar with respect to many inputs in a constant multiple of the cost of evaluating it. A library records the operations of the forward pass and replays them backwards, so a new layer needs only its own local derivative and everything above composes automatically. The hand derivation here is worth doing once, because it tells you which of a library’s error messages are about shapes and which are about your model.

Initialization stopped being arbitrary. The vanishing-gradient exercise (Exercise 8.1) and Section 9 show that the scale of the initial weights decides whether a deep network trains at all: too small and the signal shrinks going forward, too large and sigmoid units saturate or ReLU gradients explode. The standard schemes — Glorot and Bengio’s for sigmoid and tanh units [17, Sec. 4.2] and He initialization (Definition 9.2) for ReLU — set the variance of each layer’s weights from its width, so that the size of the activations is roughly preserved from layer to layer. The experiment in Section 8 scales its weights by $1/\sqrt{\text{fan-in} + 1}$ for the same reason, so that it measures depth rather than one fixed scale. For ReLU that scale still shrinks the signal, and the factor of $\sqrt{2}$ in He initialization holds it steady.

Depth is where this chapter stops. Every network here is a stack of fully connected layers. The architectures that made deep learning work — convolutional layers for images, attention for sequences, residual connections that give the gradient a path around the activation functions entirely — are all answers to questions this chapter raises. Section 11 works through residual connections, Convolutional and Recurrent Networks builds convolutional and recurrent layers, and Attention and Transformers covers attention.

16 Further reading

- Goodfellow et al. [21], Section 6.5
- Glorot and Bengio [17] — why deep sigmoid networks were hard to train
- Starmer [29], on backpropagation, with the chain rule worked through in pictures

Watch

Andrew Ng’s lectures for DeepLearning.AI work through backpropagation from computation graphs: the computation graph [30]; derivatives on the graph [31]; the intuition for a network [32]; the forward and backward passes of a deep network [33]; vanishing and exploding gradients [34]; and checking a gradient numerically [35].

- [1] G. Sanderson, “Neural networks.” 3Blue1Brown, video series on YouTube; chapters 5 and 6 added in 2024, 2017. Available: https://www.youtube.com/playlist?list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi
- [2] J. Starmer, “Neural networks pt. 2: Backpropagation main ideas.” StatQuest, video on YouTube, 2020. Available: <https://www.youtube.com/watch?v=IN2XmBhILt4>
- [3] B. Hutchinson, “DATA 471/571 (machine learning) and CSCI 481/581 (deep learning) video lectures.” Western Washington University. Accessed: Sep. 28, 2026. [Online]. Available: https://facultyweb.cs.wvu.edu/~hutchib2/video_lectures/data371/
- [4] GeeksforGeeks, “Backpropagation in neural network.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/backpropagation-in-neural-network/>
- [5] J. Starmer, “The chain rule, clearly explained.” StatQuest, video on YouTube, 2020. Available: <https://www.youtube.com/watch?v=w1lmyxrtQHQ>
- [6] J. Starmer, “Backpropagation details pt. 1: Optimizing 3 parameters simultaneously.” StatQuest, video on YouTube, 2020. Available: <https://www.youtube.com/watch?v=iyn2zdALii8>
- [7] J. Starmer, “Neural networks part 7: Cross entropy derivatives and backpropagation.” StatQuest, video on YouTube, 2021. Available: <https://www.youtube.com/watch?v=xBEh66V9gZo>
- [8] J. Starmer, “The SoftMax derivative, step-by-step.” StatQuest, video on YouTube, 2021. Available: <https://www.youtube.com/watch?v=M59JEIEPgIg>
- [9] Wikipedia contributors, “Backpropagation — Wikipedia, the free encyclopedia.” 2026. Available: <https://en.wikipedia.org/wiki/Backpropagation>
- [10] J. Starmer, “Backpropagation details pt. 2: Going bonkers with the chain rule.” StatQuest, video on YouTube, 2020. Available: <https://www.youtube.com/watch?v=GKZoOHXGcLo>
- [11] F.-F. Li *et al.*, “CS231N: Deep learning for computer vision, spring 2025.” Stanford Online, lecture videos on YouTube, 2025. Available: <https://www.youtube.com/playlist?list=PLoROMvodv4rOmsNzYBMe0gJY2XS8AQg16>
- [12] A. Ng *et al.*, “CS229: Machine learning, autumn 2018.” Stanford Online, lecture videos on YouTube, 2018. Available: <https://www.youtube.com/playlist?list=PLoROMvodv4rMiGQp3WXShtMGgzqpfVfbU>
- [13] GeeksforGeeks, “Vanishing and exploding gradients problems in deep learning.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/deep-learning/vanishing-and-exploding-gradients-problems-in-deep-learning/>
- [14] GeeksforGeeks, “ReLU activation function in deep learning.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/deep-learning/relu-activation-function-in-deep-learning/>
- [15] K. He, X. Zhang, S. Ren, and J. Sun, “Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification,” in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034. Available: <https://arxiv.org/abs/1502.01852>
- [16] Wikipedia contributors, “Vanishing gradient problem — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Vanishing_gradient_problem
- [17] X. Glorot and Y. Bengio, “Understanding the difficulty of training deep feedforward neural networks,” in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256. Available: <https://proceedings.mlr.press/v9/glorot10a.html>
- [18] GeeksforGeeks, “Weight initialization techniques for deep neural networks.” GeeksforGeeks, tutorial article, 2025. Available: <https://www.geeksforgeeks.org/machine-learning/weight-initialization-techniques-for-deep-neural-networks/>
- [19] R. Pascanu, T. Mikolov, and Y. Bengio, “On the difficulty of training recurrent neural networks,” in *Proceedings of the 30th international conference on machine learning*, 2013, pp. 1310–1318. Available: <https://arxiv.org/abs/1211.5063>

- [20] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An introduction to statistical learning with applications in R*, 2nd ed. Springer, 2021. Available: <https://www.statlearning.com/>
- [21] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2016. Available: <https://www.deeplearningbook.org/>
- [22] Wikipedia contributors, “Rectified linear unit — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Rectified_linear_unit
- [23] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778. doi: 10.1109/CVPR.2016.90¹.
- [24] Wikipedia contributors, “Residual neural network — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Residual_neural_network
- [25] D. Balduzzi, M. Frean, L. Leary, J. P. Lewis, K. W.-D. Ma, and B. McWilliams, “The shattered gradients problem: If resnets are the answer, then what is the question?” in *Proceedings of the 34th international conference on machine learning*, 2017, pp. 342–350. Available: <https://arxiv.org/abs/1702.08591>
- [26] J. Starmer, “Essential matrix algebra for neural networks, clearly explained.” StatQuest, video on YouTube, 2023. Available: <https://www.youtube.com/watch?v=ZTt9gsGcdDo>
- [27] F. Pedregosa *et al.*, “Scikit-learn: Machine learning in Python,” *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011, Available: <https://jmlr.org/papers/v12/pedregosa11a.html>
- [28] GeeksforGeeks, “What is saliency map?” GeeksforGeeks, tutorial article, 2021. Available: <https://www.geeksforgeeks.org/machine-learning/what-is-saliency-map/>
- [29] J. Starmer, *The StatQuest illustrated guide to neural networks and AI: With hands-on examples in PyTorch*. Independently published, 2025. Available: <https://statquest.org/statquest-store/>
- [30] A. Ng, “Computation graph.” DeepLearning.AI, Deep Learning Specialization, Course 1; video on YouTube, 2017. Available: <https://www.youtube.com/watch?v=hCP1vGoCdYU>
- [31] A. Ng, “Derivatives with computation graphs.” DeepLearning.AI, Deep Learning Specialization, Course 1; video on YouTube, 2017. Available: <https://www.youtube.com/watch?v=nJyUyKN-XBQ>
- [32] A. Ng, “Backpropagation intuition.” DeepLearning.AI, Deep Learning Specialization, Course 1; video on YouTube, 2017. Available: <https://www.youtube.com/watch?v=yXcQ4B-YSjQ>
- [33] A. Ng, “Forward and backward propagation.” DeepLearning.AI, Deep Learning Specialization, Course 1; video on YouTube, 2017. Available: <https://www.youtube.com/watch?v=qzPQ8cEsVK8>
- [34] A. Ng, “Vanishing/exploding gradients.” DeepLearning.AI, Deep Learning Specialization, Course 2; video on YouTube, 2017. Available: <https://www.youtube.com/watch?v=qhXZsFVxGKo>
- [35] A. Ng, “Gradient checking.” DeepLearning.AI, Deep Learning Specialization, Course 2; video on YouTube, 2017. Available: <https://www.youtube.com/watch?v=QrzApibhohY>

¹<https://doi.org/10.1109/CVPR.2016.90>