

Gradient Descent

Last modified: 2026-10-28

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

i Source

Consolidated from two sources, each of which introduces gradient descent as a digression within its logistic regression material: Kameron Decker Harris' 2022 handwritten lecture notes, whose Week 6 contains most of it, and Brian Hutchinson's Fall 2025 lecture notes, reproduced in Logistic Regression, which is the fuller of the two on line search — it is where both hyperparameters $b \in (0, 1)$ and $c_1 \in (0, 0.5]$ come from. Two separate copies of one digression is the argument for pulling it out: many models after Linear Regression are fitted this way, so the optimizer deserves its own treatment rather than being re-derived in each chapter that needs it.

The source notes' regularized stochastic step is corrected in Solution 10.17; the correction is flagged there rather than applied silently.

1 Why an iterative method

Linear Regression computes its fitted model¹ from a closed-form solution². Set the gradient of the squared-error cost to zero, and the resulting normal equations are linear in $\vec{\beta}$. When $X^\top X$ is invertible, one matrix solve produces the exact minimizer, the closed form $\hat{\vec{\beta}} = (X^\top X)^{-1} X^\top \tilde{y}$. Ridge regression has a closed-form solution of the same kind. Few other models in this course have one.

These models³ have costs whose minimizer has no closed-form solution⁴:

- the lasso;
- logistic regression;
- support vector machines;
- neural networks.

What we have instead is the ability to evaluate the cost at a point, and to evaluate its gradient there. Gradient descent is the algorithm that turns those two abilities into a fit.

Exercise 1.1 (Why we cannot always just solve for $\vec{\beta}$). Least squares has a closed-form solution: set the gradient to zero, and the resulting normal equations $(D^\top D)\vec{\beta} = D^\top \tilde{y}$ are linear in $\vec{\beta}$, so when $D^\top D$ is invertible^a one matrix solve gives the answer. Say precisely which feature of the squared-error cost makes its gradient linear in $\vec{\beta}$.

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

¹<https://morrison-lab.github.io/sds/model-validation.html#def-fitted-model>

²<https://morrison-lab.github.io/mds/expressions-algorithms.html#def-closed-form-solution>

³<https://morrison-lab.github.io/sds/estimation.html#def-model>

⁴<https://morrison-lab.github.io/mds/expressions-algorithms.html#def-closed-form-solution>

Solution

Solution 1.1. The squared-error cost is a *quadratic* in $\vec{\beta}$: every appearance of $\vec{\beta}$ is inside $\vec{\beta} \cdot \tilde{x}_i$, and that term is squared and nothing more. Differentiating a quadratic drops the degree by one, so the gradient is affine in $\vec{\beta}$, and setting an affine function to zero is a linear system.

Exercise 1.2 (Why we cannot always just solve for $\vec{\beta}$ (part 2)). Use the setup of Exercise 1.1. Logistic regression minimizes

$$C(\vec{\beta}) = \sum_{i=1}^n \left[-y_i \log \text{expit}(\vec{\beta} \cdot \tilde{x}_i) - (1 - y_i) \log(1 - \text{expit}(\vec{\beta} \cdot \tilde{x}_i)) \right]$$

where $\text{expit}(z) = 1/(1 + e^{-z})$ is the logistic (sigmoid^a) function, which logistic regression introduces in full. The gradient is $\nabla C = \sum_{i=1}^n (\text{expit}(\vec{\beta} \cdot \tilde{x}_i) - y_i) \tilde{x}_i$. Set that to zero and explain why the same step fails.

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

Solution

Solution 1.2. The score equations are

$$\sum_{i=1}^n (\text{expit}(\vec{\beta} \cdot \tilde{x}_i) - y_i) \tilde{x}_i = \vec{0}$$

The unknown $\vec{\beta}$ sits inside $\text{expit}(z) = 1/(1 + e^{-z})$, so each equation is a sum of exponentials of $\vec{\beta}$ rather than a linear combination^a of its entries. There is no rearrangement that isolates $\vec{\beta}$ on one side: the system is non-linear, and $p + 1$ non-linear equations in $p + 1$ unknowns have no general solution formula. The cost is still convex^b, which rules out spurious local minima, but convexity alone does not deliver a minimizer. The logistic loss is strictly convex in the fitted scores $D\vec{\beta}$ rather than in $\vec{\beta}$, so uniqueness needs D to have full column rank^c, and existence needs the classes not to be perfectly separable — otherwise the cost keeps falling as $\|\vec{\beta}\|$ grows and no finite minimizer exists at all. Given both, the minimizer is unique, and we simply have no way to write it down. Finding it is a search, which is the subject of the rest of this chapter.

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

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

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

Exercise 1.3 (Why we cannot always just solve for $\vec{\beta}$ (part 3)). Use the setup of Exercise 1.1. Give a second reason to fit iteratively that has nothing to do with non-linearity, and applies even to plain least squares.

Solution

Solution 1.3. Size. Forming $D^\top D$ costs $O(np^2)$ and inverting it costs $O(p^3)$, and both require the whole of D to be available at once. With p in the thousands the cube dominates, and with n in the hundreds of millions the data does not fit in memory at all. A gradient step needs only $O(np)$ work — or, as Solution 10.5 shows, only $O(p)$ — and can be taken while the data streams past. So an iterative method is the answer both to *we cannot solve it* and to *we cannot afford to solve it*.

i Watch

Andrew Ng introduces gradient descent in the second CS229 lecture, from 18:06^a [1], Lecture 2.

^ahttps://www.youtube.com/watch?v=4b4MUYve_U8&t=1086s

Exercise 1.4 (Is an argmin an algorithm?). A squared-error objective is an expression^a in $\vec{\beta}$, and gradient descent^b is an algorithm^c that minimizes it. *Math for Data Science*^d asks how the two differ for a polynomial^e.

Now take a general cost function $C(\vec{\beta})$, not only squared error. Can you turn $\arg \min_{\vec{\beta}} C(\vec{\beta})$ into a list of steps that returns the minimizing $\vec{\beta}$ exactly?

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

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

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

^d<https://morrison-lab.github.io/mds/expressions-algorithms.html#exr-expression-or-algorithm>

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

Solution

Solution 1.4. Not in general. Only for special costs, such as the squared error of linear regression, does a formula exist, and evaluating that formula is an algorithm^a. For other costs, such as the logistic cost of Exercise 1.2, the argmin^b is a well-defined value that no formula returns. We choose an algorithm that approximates it.

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

^b<https://morrison-lab.github.io/mds/algebra-basics.html#def-argmin>

Example 1.1 (An argmin with no closed-form expression). The logistic cost $C(\vec{\beta})$ of Exercise 1.2 is an expression^a, and $\hat{\vec{\beta}} = \arg \min_{\vec{\beta}} C(\vec{\beta})$ is a well-defined vector, but no closed-form expression^b returns it. Gradient descent is an algorithm^c that supplies steps approaching it.

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

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

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

2 The descent direction

At a given $\vec{\beta}_t$, we want to move so that the cost goes down. The gradient

$$\nabla_{\vec{\beta}} C(\vec{\beta}) = \begin{bmatrix} \partial C / \partial \beta_0 \\ \partial C / \partial \beta_1 \\ \vdots \\ \partial C / \partial \beta_p \end{bmatrix}$$

collects the partial derivatives, so it says how fast C rises in each coordinate direction. It also says more than that: it identifies the single direction of fastest increase, and therefore its negative is the direction of fastest decrease.

Exercise 2.1 (Show that the negative gradient is the steepest direction). “Go downhill” does not by itself say *which* way. This exercise shows that the negative gradient is the best answer, and says what “best” means here.

Write the first-order Taylor expansion of $C(\vec{\beta} + \varepsilon \underline{u})$ about $\vec{\beta}$, for a small scalar $\varepsilon > 0$ and a direction $\underline{u}$ with $\|\underline{u}\|_2 = 1$.

Solution

Solution 2.1.

$$C(\vec{\beta} + \varepsilon \tilde{u}) = C(\vec{\beta}) + \varepsilon \tilde{u} \cdot \nabla C(\vec{\beta}) + O(\varepsilon^2)$$

Exercise 2.2 (Show that the negative gradient is the steepest direction (part 2)). Use the setup of Exercise 2.1.

Which term in that expansion depends on $\tilde{u}$? Making the step as downhill as possible therefore means^a minimizing that term over all unit $\tilde{u}$.

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

Solution

Solution 2.2. Only the middle term. $C(\vec{\beta})$ is fixed at the current point, and the remainder shrinks faster than ε as the step shrinks, so for a short enough step the sign and size of the change are decided by $\tilde{u} \cdot \nabla C$ alone.

Exercise 2.3 (Show that the negative gradient is the steepest direction (part 3)). Use the setup of Exercise 2.1.

State the Cauchy–Schwarz inequality, including the condition for equality, and use it to find the minimizing $\tilde{u}$.

Solution

Solution 2.3. Cauchy–Schwarz says $|\tilde{a}^\top b| \leq \|\tilde{a}\|_2 \|\tilde{b}\|_2$, with equality exactly when $\tilde{a}$ and b are parallel — one is a scalar multiple of the other.

With $\|\tilde{u}\|_2 = 1$ this gives

$$-\|\nabla C\|_2 \leq \tilde{u} \cdot \nabla C \leq \|\nabla C\|_2$$

The lower bound is attained when $\tilde{u}$ is parallel to ∇C and the inner product is negative, which determines $\tilde{u}$ uniquely:

$$\tilde{u}^* = -\frac{\nabla C}{\|\nabla C\|_2}$$

and then $\tilde{u}^* \cdot \nabla C = -\|\nabla C\|_2$. This is the same geometry as Linear functions as inner products: $\tilde{u} \cdot \nabla C = \|\nabla C\|_2 \cos \theta$, minimized at $\theta = 180^\circ$.

Exercise 2.4 (Show that the negative gradient is the steepest direction (part 4)). Use the setup of Exercise 2.1.

The gradient of C at that point is $\nabla C = (3, -4)^\top$. Write the unit step direction, and say how much C falls per unit of step length, to first order.

Solution

Solution 2.4. $\|\nabla C\|_2 = \sqrt{9 + 16} = 5$, so

$$\begin{aligned} \tilde{u}^* &= -\frac{1}{5}(3, -4)^\top \\ &= (-0.6, 0.8)^\top \end{aligned}$$

and C falls at rate $\|\nabla C\|_2 = 5$ per unit of step length, to first order. Note that the *rate* is the length of the gradient, while the *direction* is the gradient normalized. Both come out of the same inequality.

Exercise 2.5 (Show that the negative gradient is the steepest direction (part 5)). Use the setup of Exercise 2.1. Name two things this argument does *not* establish.

Solution

Solution 2.5. Two things, and both matter for the rest of the chapter.

It is local. The expansion is only accurate for small ε , so “steepest” describes the slope^a at the current point and says nothing about where the step lands. A direction that starts out steepest can be overtaken within the very same step, which is exactly why the step size needs its own treatment in [Choosing the step size](#).

It depends on the norm^b. The constraint was $\|u\|_2 = 1$. Measure the unit step with a different norm and a different direction wins: under $\|u\|_\infty = 1$ the steepest direction is $-\text{sign}(\nabla C)$, which is the sign-of-the-gradient rule some optimizers use. So “the steepest descent direction” is a statement about the geometry we imposed, not a property of C on its own.

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

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

i Watch

Weinberger derives the descent direction from a local approximation of the cost, from 13:30^a [2], Lecture 12.

^a<https://www.youtube.com/watch?v=o6FfdP2uYh4&t=810s>

3 The algorithm

The math notes define gradient descent⁵, and this chapter uses it as stated there. Here the cost is C , the parameter vector is $\vec{\beta}$, and each step is scaled by α . In this notation, the update at step t is

$$\vec{\beta}_t = \vec{\beta}_{t-1} - \alpha \nabla_{\vec{\beta}} C(\vec{\beta}_{t-1}).$$

The algorithm⁶ repeats this update until it has converged. Three things in the update have to be chosen, and each gets its own section:

- where to start;
- how long a step to take;
- when to stop.

Definition 3.1 (Learning rate). The **learning rate** $\alpha > 0$ of gradient descent^a is the number that multiplies the gradient in each update. It is also called the **step size**.

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

Example 3.1 (One step at two learning rates). Let:

- $C(\beta) = \beta^2$ be the cost, with derivative $C'(\beta) = 2\beta$;
- β_0 be the starting value of β ;
- α_{small} and α_{large} be the two [learning rates](#) to compare.

Start at $\beta_0 = 2$, so $C'(\beta_0) = 4$. The first update is $\beta_1 = \beta_0 - 4\alpha$:

- for $\alpha_{\text{small}} = 0.25$, $\beta_1 = 1$, which is closer to the minimizer 0;

⁵<https://morrison-lab.github.io/mds/optimization.html#def-gradient-descent>

⁶<https://morrison-lab.github.io/mds/expressions-algorithms.html#def-algorithm>

- for $\alpha_{\text{large}} = 1.25$, $\beta_1 = -3$, which is farther from 0 than the start was. A larger **learning rate** moves farther, but too far overshoots the minimizer.

Definition 3.2 (Gradient flow). Let C be the cost. **Gradient flow** is the curve $\vec{\beta}(t)$ that solves the differential equation

$$\frac{d\vec{\beta}(t)}{dt} = -\nabla C(\vec{\beta}(t)).$$

Gradient descent^a is Euler's method for this equation, with the **learning rate** α as the step in t . The flow runs downhill, perpendicular to the level sets^b.

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

^b<https://morrison-lab.github.io/mds/vector-calculus.html#def-level-set>

Example 3.2 (Gradient descent follows the flow). Let:

- $C(\beta) = \frac{1}{2}\beta^2$ be the cost, so $\nabla C(\beta) = \beta$;
- β_0 be the starting value of β ;
- T be the time at which to compare the two paths.

The **gradient flow** solves $\frac{d\beta(t)}{dt} = -\beta(t)$, and its solution is $\beta(t) = \beta_0 \exp\{-t\}$. Gradient descent^a with **learning rate** α multiplies β by $1 - \alpha$ at each step, so after k steps it has reached time $t = k\alpha$ with $\beta_k = \beta_0(1 - \alpha)^k$.

Start at $\beta_0 = 1$ and run to time $T = 1$:

- for $\alpha = 0.5$, $\beta_2 = 0.25$;
- for $\alpha = 0.01$, $\beta_{100} = 0.366$;
- the flow gives $\beta(T) = 0.3679$.

The smaller the learning rate, the closer the descent stays to the flow.

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

Exercise 3.1 (Take one step by hand, three times). Use the three points of Solve the normal equations by hand:

$$(x_1, y_1) = (0, 1), \quad (x_2, y_2) = (1, 3), \quad (x_3, y_3) = (2, 3)$$

so that, with the intercept trick,

$$D = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 2 \end{bmatrix}, \quad \underset{\sim}{y} = \begin{bmatrix} 1 \\ 3 \\ 3 \end{bmatrix}, \quad J(\vec{\beta}) = \|D\vec{\beta} - \underset{\sim}{y}\|_2^2$$

Start from $\vec{\beta}_0 = (0, 0)^\top$ and work in exact fractions.

Compute $J(\vec{\beta}_0)$ and $\nabla J(\vec{\beta}_0)$.

Solution

Solution 3.1. At $\vec{\beta}_0 = \vec{0}$ the fitted values are all zero, so the residual is $D\vec{\beta}_0 - \underset{\sim}{y} = (-1, -3, -3)^\top$ and

$$\begin{aligned} J(\vec{\beta}_0) &= 1 + 9 + 9 \\ &= 19 \end{aligned}$$

Using $\nabla J = 2D^\top(D\vec{\beta} - \underset{\sim}{y})$ from The gradient,

$$D^T \begin{bmatrix} -1 \\ -3 \\ -3 \end{bmatrix} = \begin{bmatrix} -1 & -3 & -3 \\ 0 & -3 & -6 \end{bmatrix} = \begin{bmatrix} -7 \\ -9 \end{bmatrix}, \quad \nabla J(\vec{\beta}_0) = \begin{bmatrix} -14 \\ -18 \end{bmatrix}$$

The gradient is negative in both coordinates, so both the intercept and the slope should increase — which is right, since the data sits above and to the right of the origin.

Exercise 3.2 (Take one step by hand, three times (part 2)). Use the setup of Exercise 3.1. Take one step $\vec{\beta}_1 = \vec{\beta}_0 - \alpha \nabla J(\vec{\beta}_0)$ for each of $\alpha = \frac{1}{20}$, $\alpha = \frac{1}{10}$ and $\alpha = \frac{3}{20}$, and compute $J(\vec{\beta}_1)$ in each case.

Solution

Solution 3.2. Each step is $\vec{\beta}_1 = -\alpha(-14, -18)^T = \alpha(14, 18)^T$.

α	$\vec{\beta}_1$	fitted values	residual	$J(\vec{\beta}_1)$
1/20	(7/10, 9/10)	(7/10, 16/10, 25/10)	(-3/10, -14/10, -5/10)	2.30
1/10	(7/5, 9/5)	(7/5, 16/5, 5)	(2/5, 1/5, 2)	4.20
3/20	(21/10, 27/10)	(21/10, 48/10, 75/10)	(11/10, 18/10, 45/10)	24.70

Exercise 3.3 (Take one step by hand, three times (part 3)). Use the setup of Exercise 3.1. Solve the normal equations by hand found the minimizer $\vec{\beta}^* = (4/3, 1)^T$ with $J(\vec{\beta}^*) = 2/3$. Which of the three steps moved downhill?

Solution

Solution 3.3. The first two moved downhill: 2.30 and 4.20 are both below 19. The third went *up*, from 19 to 24.70.

Exercise 3.4 (Take one step by hand, three times (part 4)). Use the setup of Exercise 3.1. Which of the three steps got closest to the minimizer $\vec{\beta}^*$?

Solution

Solution 3.4. The closest to $\vec{\beta}^* = (4/3, 1) \approx (1.333, 1.000)$ is $\alpha = 1/20$, at (0.7, 0.9), a distance of about 0.64. The $\alpha = 1/10$ step lands at (1.4, 1.8), distance about 0.80: it has overshoot in the slope^a while the intercept^b is nearly right. The $\alpha = 3/20$ step lands at (2.1, 2.7), distance about 1.87, further from the answer than where it started. Note that J and distance-to- $\vec{\beta}^*$ order the three steps the same way here, but they are different quantities and will not always agree.

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

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

Exercise 3.5 (Take one step by hand, three times (part 5)). Use the setup of Exercise 3.1. One of the three raised the cost. Nothing about the direction was wrong. Say what went wrong instead.

Solution

Solution 3.5. The step was too long. The direction $-\nabla J$ is downhill, as Solution 2.3 proved, but only over a short enough step — the Taylor expansion that justified it drops an $O(\varepsilon^2)$ term, and at $\alpha = 3/20$ that term is no longer small. Geometrically, the step crosses the valley and ends at a higher cost than it started.

The direction is free to choose and always correct. The length is the part that has to be chosen carefully, and [Choosing the step size](#) makes the boundary exact: for this D , any $\alpha < 0.1396$ shrinks the error and any $\alpha > 0.1396$ grows it. So $3/20 = 0.15$ failed and $1/10$ did not.

Definition 3.3 (Rosenbrock function). The **Rosenbrock function** on $\mathbb{R}^2$ is

$$f(\tilde{x}) = 100(x_2 - x_1^2)^2 + (1 - x_1)^2, \quad \tilde{x} = (x_1, x_2)^\top.$$

It is a standard test problem for optimizers [3].

Example 3.3 (Values of the Rosenbrock function). Evaluate the function at the minimizer $(1, 1)$, at the origin, and at the standard starting point $(-1.2, 1)$. Then evaluate it along the curve $x_2 = x_1^2$, which the function treats as the floor of a narrow, curved valley:

3.1 Python

```
import numpy as np

def rosenbrock_rf(x1, x2):
    return 100 * (x2 - x1 ** 2) ** 2 + (1 - x1) ** 2

points_rf = {"minimizer": (1, 1), "origin": (0, 0), "start": (-1.2, 1)}
print({name: round(rosenbrock_rf(*p), 3) for name, p in points_rf.items()})

{'minimizer': 0, 'origin': 1, 'start': 24.2}

# On the curve x2 = x1^2 the first term vanishes, leaving (1 - x1)^2.
x_curve_rf = np.array([-1.0, 0.0, 0.5, 1.0, 2.0])
print(rosenbrock_rf(x_curve_rf, x_curve_rf ** 2), (1 - x_curve_rf) ** 2)

[4.  1.  0.25 0.  1. ] [4.  1.  0.25 0.  1. ]
```

3.2 R

```
rosenbrock_rf <- function(x1, x2) 100 * (x2 - x1^2)^2 + (1 - x1)^2

points_rf <- list(minimizer = c(1, 1), origin = c(0, 0), start = c(-1.2, 1))
sapply(points_rf, function(p) round(rosenbrock_rf(p[1], p[2]), 3))

minimizer    origin    start
      0.0       1.0     24.2

# On the curve x2 = x1^2 the first term vanishes, leaving (1 - x1)^2.
x_curve_rf <- c(-1, 0, 0.5, 1, 2)
rbind(on_curve = rosenbrock_rf(x_curve_rf, x_curve_rf^2), formula = (1 - x_curve_rf)^2)

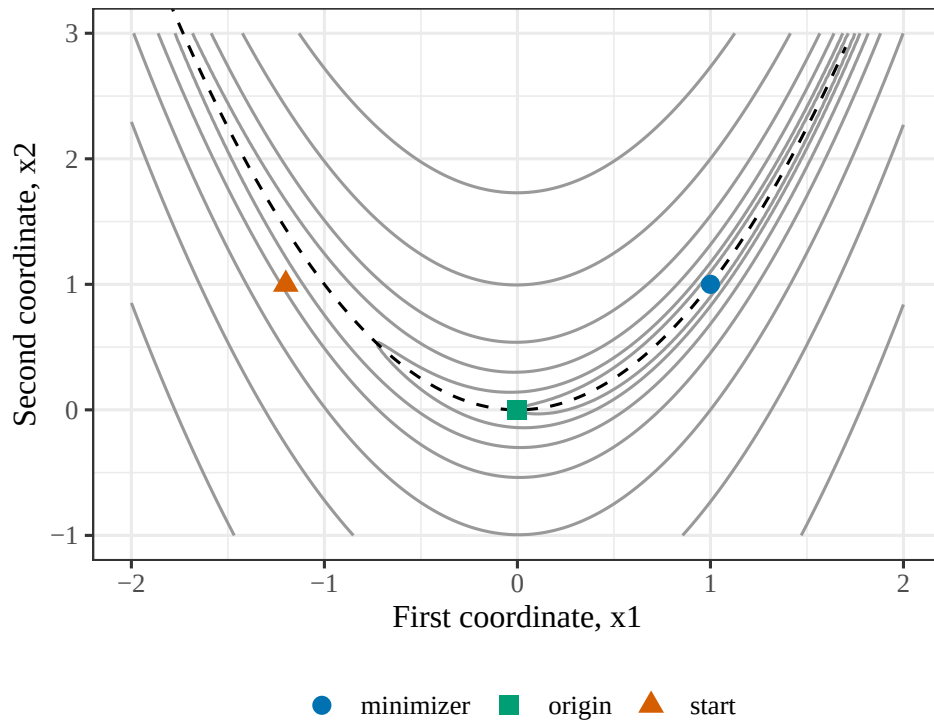
      [,1] [,2] [,3] [,4] [,5]
on_curve  4   1 0.25  0   1
formula   4   1 0.25  0   1
```

The function is 0 at $(1, 1)$ and positive everywhere else, so $(1, 1)$ is the global minimizer. At the starting point the value is 24.2, mostly from the first term, because $(-1.2, 1)$ is off the curve $x_2 = x_1^2$. On the curve the function is small and varies slowly, so the valley floor is nearly flat even though its walls are

steep.

3.3 R

```
library(ggplot2)
rf_grid <- expand.grid(x1 = seq(-2, 2, length.out = 201), x2 = seq(-1, 3, length.out = 201))
rf_grid$f <- rosenbrock_rf(rf_grid$x1, rf_grid$x2)
rf_points <- data.frame(
  name = names(points_rf),
  x1 = sapply(points_rf, `[`, 1),
  x2 = sapply(points_rf, `[`, 2)
)
rf_curve <- data.frame(x1 = seq(-2, 1.7, length.out = 200))
rf_curve$x2 <- rf_curve$x1^2
ggplot(rf_grid, aes(x1, x2)) +
  geom_contour(aes(z = f), colour = "grey60", breaks = c(1, 3, 10, 30, 100, 300, 1000)) +
  geom_line(data = rf_curve, linetype = "dashed") +
  geom_point(aes(colour = name, shape = name), data = rf_points, size = 3) +
  scale_colour_manual(values = c(minimizer = "#0072B2", origin = "#009E73", start = "#D55E00")) +
  scale_shape_manual(values = c(minimizer = 16, origin = 15, start = 17)) +
  coord_cartesian(xlim = c(-2, 2), ylim = c(-1, 3)) +
  labs(x = "First coordinate, x1", y = "Second coordinate, x2", colour = NULL, shape = NULL)
```



(a)

3.4 Python

```
import matplotlib.pyplot as plt
import numpy as np

g1_rf, g2_rf = np.meshgrid(np.linspace(-2, 2, 201), np.linspace(-1, 3, 201))
fig, ax = plt.subplots(figsize=(5, 4), layout="constrained")
ax.contour(g1_rf, g2_rf, rosenbrock_rf(g1_rf, g2_rf), levels=[1, 3, 10, 30, 100, 300, 1000], colors="grey")
t_rf = np.linspace(-2, 1.7, 200)
ax.plot(t_rf, t_rf ** 2, color="black", linestyle="--");
style_rf = {"minimizer": ("#0072B2", "o"), "origin": ("#009E73", "s"), "start": ("#D55E00", "^")}
for name, p in points_rf.items():
    ax.plot(*p, color=style_rf[name][0], marker=style_rf[name][1], linestyle="", markersize=8, label=name)
ax.set_xlabel("First coordinate, x1");
ax.set_ylabel("Second coordinate, x2");
```

Exercise 3.6 (Descend the Rosenbrock function). Use Definition 3.3. Compute the gradient $\nabla f(\tilde{x})$.

Solution

Solution 3.6. Differentiate each term with the chain rule. The first term depends on both coordinates:

$$\begin{aligned}\frac{\partial}{\partial x_1} 100(x_2 - x_1^2)^2 &= 200(x_2 - x_1^2) \cdot (-2x_1) \\ &= -400 x_1 (x_2 - x_1^2),\end{aligned}$$

$$\frac{\partial}{\partial x_2} 100(x_2 - x_1^2)^2 = 200(x_2 - x_1^2).$$

The second term depends only on x_1 : $\frac{\partial}{\partial x_1} (1 - x_1)^2 = -2(1 - x_1)$. So

$$\nabla f(\tilde{x}) = \begin{bmatrix} -400 x_1 (x_2 - x_1^2) - 2(1 - x_1) \\ 200(x_2 - x_1^2) \end{bmatrix}.$$

Exercise 3.7 (Descend the Rosenbrock function (part 2)). Use the setup of Exercise 3.6. Write the gradient descent^a update $\tilde{x}_{k+1}$ with step size α as a pair of formulas in $x_{k,1}$ and $x_{k,2}$.

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

Solution

Solution 3.7. Subtracting α times the gradient from $\tilde{x}_k$, coordinate by coordinate:

$$\begin{aligned}x_{k+1,1} &= x_{k,1} + \alpha(400 x_{k,1} (x_{k,2} - x_{k,1}^2) + 2(1 - x_{k,1})), \\ x_{k+1,2} &= x_{k,2} - 200 \alpha (x_{k,2} - x_{k,1}^2).\end{aligned}$$

Exercise 3.8 (Descend the Rosenbrock function (part 3)). Use the setup of Exercise 3.6. Check the gradient numerically at $\tilde{x} = (-1.2, 1)^\top$ against a finite-difference approximation of each partial derivative.

Solution

Solution 3.8.

```
# The Rosenbrock function and its gradient from part 1.
rosenbrock <- function(x) {
  100 * (x[2] - x[1]^2)^2 + (1 - x[1])^2
}
rosenbrock_grad <- function(x) {
  c(
    -400 * x[1] * (x[2] - x[1]^2) - 2 * (1 - x[1]),
    200 * (x[2] - x[1]^2)
  )
}

# Central finite differences, one coordinate at a time, as an
# independent check on the formula: it does not use the derivation.
x0 <- c(-1.2, 1)
eps <- 1e-6
fd_grad <- vapply(
  1:2,
  function(j) {
    e <- replace(numeric(2), j, eps)
    (rosenbrock(x0 + e) - rosenbrock(x0 - e)) / (2 * eps)
  },
  numeric(1)
)
grad_gap <- max(abs(fd_grad - rosenbrock_grad(x0)))
```

The largest gap between the formula and the finite differences is 2.2e-08, which is finite-difference error, so the gradient agrees with the numerical derivative.

Exercise 3.9 (Descend the Rosenbrock function (part 4)). Use the setup of Exercise 3.6. Run 1000 steps with $\alpha = 0.001$ from $\tilde{x}_0 = (-1.2, 1)^\top$.

Solution

Solution 3.9.

```
# 1000 gradient descent steps with a fixed step size h.
h <- 0.001
x <- x0
for (k in 1:1000) {
  x <- x - h * rosenbrock_grad(x)
}
x
[1] 0.3272628 0.1040128
```

Exercise 3.10 (Descend the Rosenbrock function (part 5)). Use the setup of Exercise 3.6. Is the final iterate close to the minimizer of f ?

Solution

Solution 3.10. The function is zero at $(1, 1)^\top$ and positive everywhere else, so $(1, 1)^\top$ is the minimizer. After 1000 steps the iterate is $(0.327, 0.104)^\top$ and f has fallen from 24.2 to 0.454. That is progress, but the iterate is not close to $(1, 1)^\top$. The minimizer sits at the end of a long, curved, narrow valley, the same shape that Solution 7.15 describes for a badly conditioned least squares problem, and a step size

small enough not to bounce off the valley walls advances only slowly along its floor.

Watch

Andrew Ng writes the algorithm out in the second CS229 lecture, from 33:01^a [1], Lecture 2.

^ahttps://www.youtube.com/watch?v=4b4MUyve_U8&t=1981s

Watch

Starmer works through gradient descent step by step, from 1:25^a [4].

^a<https://www.youtube.com/watch?v=sDv4fs2SB8&t=85s>

Watch

Hutchinson covers gradient descent, step size and learning rate^a (15 min) [5]. The video site requires a login, which enrolled students find in the syllabus.

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

Read

GeeksforGeeks explains gradient descent^a [6].

^a<https://www.geeksforgeeks.org/machine-learning/gradient-descent-algorithm-and-its-variants/>

Read

The University of Washington's CSE 446 slides cover gradient descent^a, convexity^b and the theory of gradient descent^c [7], Lectures 9 to 11.

^a<https://courses.cs.washington.edu/courses/cse446/26sp/schedule/lectures/09-gradient%20descent.pdf>

^b<https://courses.cs.washington.edu/courses/cse446/26sp/schedule/lectures/10-convexity.pdf>

^c<https://courses.cs.washington.edu/courses/cse446/26sp/schedule/lectures/11-gradient%20descent%20theory.pdf>

4 Convexity and stationary points

Whether descent can be trusted depends on the shape of the cost J . The definitions of a convex⁷ function, a stationary point⁸, a local minimizer⁹, a local maximizer¹⁰ and a saddle point¹¹ are in the math notes, and are used here as they are stated there. This section adds an example for each idea.

Example 4.1. Take the one-parameter squared-error cost $J(\beta) = (\beta - 3)^2$, and the points $\beta_1 = 0$ and $\beta_2 = 4$ with $t = 1/2$. The midpoint is $\beta = 2$, and

⁷<https://morrison-lab.github.io/mds/algebra-sums.html#def-convex-function>

⁸<https://morrison-lab.github.io/mds/optimization.html#def-stationary-point>

⁹<https://morrison-lab.github.io/mds/algebra-basics.html#def-local-minimizer>

¹⁰<https://morrison-lab.github.io/mds/optimization.html#def-local-maximizer>

¹¹<https://morrison-lab.github.io/mds/optimization.html#def-saddle-point>

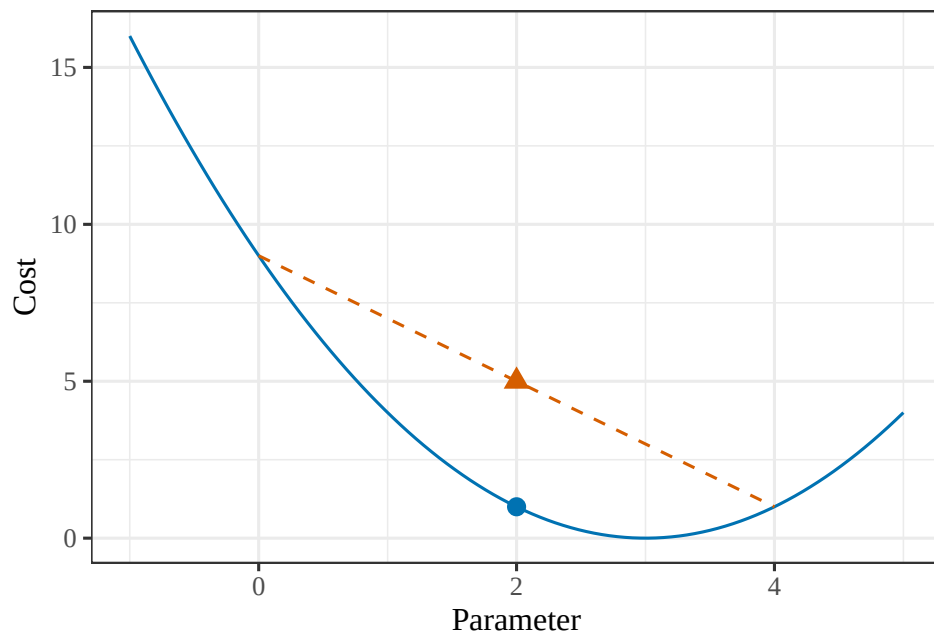
$$\begin{aligned}
J(2) &= 1 \\
&\leq \tfrac{1}{2}J(0) + \tfrac{1}{2}J(4) \\
&= \tfrac{1}{2}(9) + \tfrac{1}{2}(1) \\
&= 5.
\end{aligned}$$

The surface at the midpoint lies below the chord, as the definition of a convex^{[a](#)} function requires. A single check at one pair of points cannot prove convexity, but it can disprove it, as the next cost shows. By contrast, $J(\beta) = -\beta^2$ is not convex: at $\beta_1 = -1$, $\beta_2 = 1$ and $t = 1/2$, $J(0) = 0$ is larger than $\frac{1}{2}(-1) + \frac{1}{2}(-1) = -1$, so the chord passes below the surface.

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

4.1 R

```
library(ggplot2)
cv_cost <- function(beta) (beta - 3)^2
cv_ends <- c(0, 4)
cv_t <- 1 / 2
cv_mid <- cv_t * cv_ends[1] + (1 - cv_t) * cv_ends[2]
cv_points <- data.frame(
  beta = c(cv_mid, cv_mid),
  cost = c(cv_cost(cv_mid), cv_t * cv_cost(cv_ends[1]) + (1 - cv_t) * cv_cost(cv_ends[2])),
  where = c("surface", "chord")
)
ggplot() +
  geom_function(fun = cv_cost, colour = "#0072B2", xlim = c(-1, 5)) +
  geom_line(
    data = data.frame(beta = cv_ends, cost = cv_cost(cv_ends)),
    aes(beta, cost), colour = "#D55E00", linetype = "dashed"
  ) +
  geom_point(data = cv_points, aes(beta, cost, colour = where, shape = where), size = 3) +
  scale_colour_manual(values = c(surface = "#0072B2", chord = "#D55E00")) +
  scale_shape_manual(values = c(surface = 16, chord = 17)) +
  labs(x = "Parameter", y = "Cost", colour = NULL, shape = NULL)
```



▲ chord ● surface

(a)

4.2 Python

```
import matplotlib.pyplot as plt
import numpy as np

def cv_cost(beta):
    return (beta - 3) ** 2

cv_ends = np.array([0, 4])
cv_t = 1 / 2
cv_mid = cv_t * cv_ends[0] + (1 - cv_t) * cv_ends[1]
grid_cv = np.linspace(-1, 5, 241)
fig, ax = plt.subplots(figsize=(5, 4), layout="constrained")
```

Example 4.2. Let $J(\beta) = \beta^4 - 2\beta^2$. Its derivative is $J'(\beta) = 4\beta^3 - 4\beta = 4\beta(\beta - 1)(\beta + 1)$, so the stationary points^a are $\beta = -1, 0$ and 1 .

- $J(\pm 1) = -1$, and $J(\beta) = \beta^2(\beta^2 - 2) > -1$ for every other β near ± 1 , so both are local minima (local minimizers^b). They are in fact global minima.
- $J(0) = 0$, and $J(\beta) < 0$ for every nonzero β with $\beta^2 < 2$, so $\beta = 0$ is a local maximizer^c.

With a small step size, descent started at $\beta_0 = 0.1$ runs away from 0 and settles at $\beta = 1$, while a start at $\beta_0 = -0.1$ settles at $\beta = -1$. One equation, $J' = 0$, produced three answers of two kinds.

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

^b<https://morrison-lab.github.io/mds/algebra-basics.html#def-local-minimizer>

^c<https://morrison-lab.github.io/mds/optimization.html#def-local-maximizer>

Example 4.3. The cost $J(z_1, z_2) = 3z_1^2 + 4z_1z_2 + z_2^2$ of Linear Regression has $\nabla J = (6z_1 + 4z_2, 4z_1 + 2z_2)$, which vanishes only at the origin. Along the z_1 axis, $J(t, 0) = 3t^2 > 0$. Along the line $z_2 = -2z_1$,

$$\begin{aligned} J(t, -2t) &= 3t^2 - 8t^2 + 4t^2 \\ &= -t^2 \\ &< 0. \end{aligned}$$

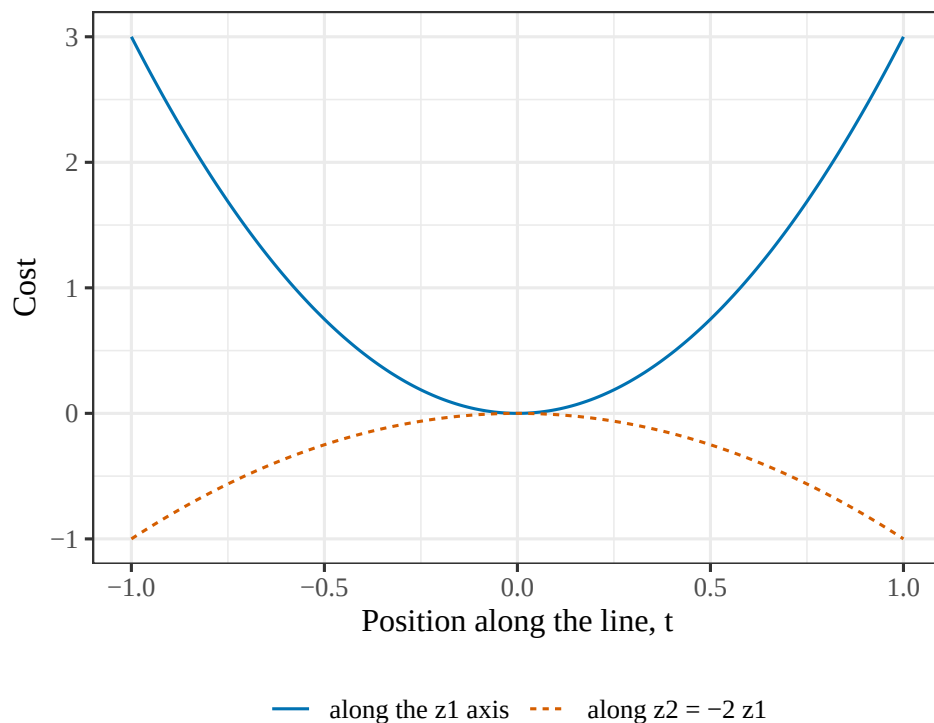
Every neighborhood of the origin holds points where J is larger and points where it is smaller, so the origin is neither a local minimum nor a local maximum. It is a saddle point^a.

A critical point where the surface flattens without turning around, such as $\beta = 0$ for $J(\beta) = \beta^3$, is a saddle point under this definition too, though it curves in only one direction.

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

4.3 R

```
library(ggplot2)
sp_cost <- function(z1, z2) 3 * z1^2 + 4 * z1 * z2 + z2^2
sp <- data.frame(t = seq(-1, 1, length.out = 201))
sp$axis <- sp_cost(sp$t, 0)
sp$line <- sp_cost(sp$t, -2 * sp$t)
sp_long <- data.frame(
  t = rep(sp$t, 2),
  cost = c(sp$axis, sp$line),
  path = rep(c("along the z1 axis", "along z2 = -2 z1"), each = nrow(sp))
)
ggplot(sp_long, aes(t, cost, colour = path, linetype = path)) +
  geom_line() +
  scale_colour_manual(values = c("#0072B2", "#D55E00")) +
  labs(x = "Position along the line, t", y = "Cost", colour = NULL, linetype = NULL)
```



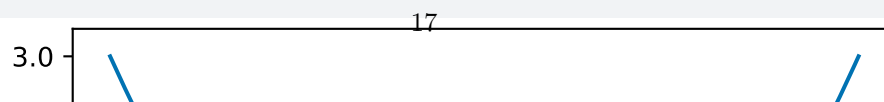
(a)

4.4 Python

```
import matplotlib.pyplot as plt
import numpy as np

def sp_cost(z1, z2):
    return 3 * z1 ** 2 + 4 * z1 * z2 + z2 ** 2

t_sp = np.linspace(-1, 1, 201)
fig, ax = plt.subplots(figsize=(5, 4), layout="constrained")
ax.plot(t_sp, sp_cost(t_sp, 0), color="#0072B2", linestyle="-", label="along the z1 axis");
ax.plot(t_sp, sp_cost(t_sp, -2 * t_sp), color="#D55E00", linestyle="--", label="along z2 = -2 z1");
ax.set_xlabel("Position along the line, t");
ax.set_ylabel("Cost");
fig.legend(loc="outside lower center", ncol=2);
plt.show()
```



5 Non-convex costs

On a non-convex cost the answer is largely decided by $\vec{\beta}_0$, before a single step is taken. The next definition names the set of starting points that reach one minimum.

Definition 5.1 (Basin of attraction). The **basin of attraction** of a local minimum^a is the set of starting points $\vec{\beta}_0$ from which the descent iterates, for a given step size, converge to that minimum [8].

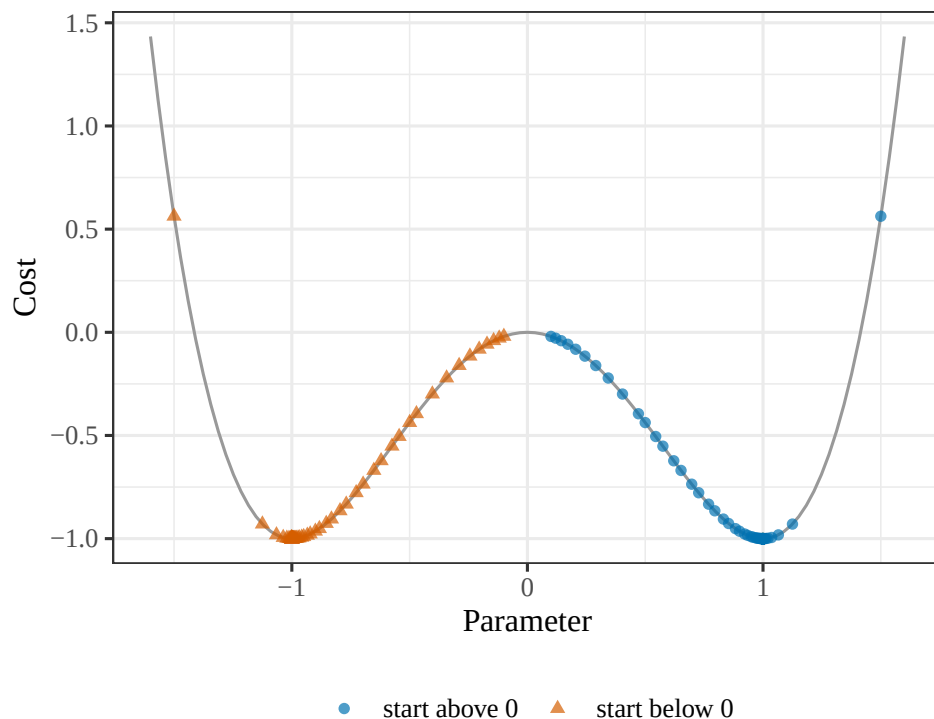
^a<https://morrison-lab.github.io/mds/algebra-basics.html#def-local-minimizer>

Example 5.1. Take $J(\beta) = \beta^4 - 2\beta^2$ again, from Example 4.2. With a small step size, a start $\beta_0 > 0$ of moderate size converges to the local minimum at $\beta = 1$, and a start $\beta_0 < 0$ of moderate size converges to the one at $\beta = -1$. So the basin of attraction of $\beta = 1$ is close to $(0, \infty)$, and that of $\beta = -1$ is close to $(-\infty, 0)$. The single start $\beta_0 = 0$ is the local maximum, where the gradient is zero and descent never moves. Which minimum the run reaches was settled by the sign of β_0 , before the first step.

The word “close” is needed because the basin depends on the step size. The curvature $J''(\beta) = 12\beta^2 - 4$ grows without bound, so for any fixed step size a start far enough from the origin overshoots and diverges. The exact basin is a bounded stretch of each half-line, and it shrinks as the step size grows.

5.1 R

```
library(ggplot2)
bw_cost <- function(theta) theta^4 - 2 * theta^2
bw_step <- 0.05
bw_starts <- c(-1.5, -0.5, -0.1, 0.1, 0.5, 1.5)
bw_iterates <- do.call(rbind, lapply(bw_starts, function(start) {
  theta <- numeric(31)
  theta[1] <- start
  for (i in 2:31) theta[i] <- theta[i - 1] - bw_step * (4 * theta[i - 1]^3 - 4 * theta[i - 1])
  data.frame(start = start, theta = theta, basin = ifelse(start > 0, "start above 0", "start below 0"))
}))
bw_iterates$cost <- bw_cost(bw_iterates$theta)
ggplot(bw_iterates, aes(theta, cost)) +
  geom_function(fun = bw_cost, colour = "grey60", xlim = c(-1.6, 1.6)) +
  geom_point(aes(colour = basin, shape = basin), alpha = 0.7) +
  scale_colour_manual(values = c("start above 0" = "#0072B2", "start below 0" = "#D55E00")) +
  labs(x = "Parameter", y = "Cost", colour = NULL, shape = NULL)
```



(a)

5.2 Python

```
import matplotlib.pyplot as plt
import numpy as np

def bw_cost(theta):
    return theta ** 4 - 2 * theta ** 2

bw_step = 0.05
fig, ax = plt.subplots(figsize=(5, 4), layout="constrained")
grid_bw = np.linspace(-1.6, 1.6, 321)
ax.plot(grid_bw, bw_cost(grid_bw), color="grey");
for start in [-1.5, -0.5, -0.1, 0.1, 0.5, 1.5]:
    theta = [start]
    for _ in range(30):
        theta.append(theta[-1] - bw_step * (4 * theta[-1] ** 3 - 4 * theta[-1]))
    theta = np.array(theta)
```

The basins of attraction (Definition 5.1) cover the parameter space, apart from a usually negligible set of starting points that reach no minimum. Level sets¹⁵ are the usual way to see them in two dimensions: the set of $\vec{\beta}$ where J takes a constant value, drawn for a few values of that constant. For the linear function of Linear Regression every level set is a hyperplane, and the picture carries little information. For a non-convex risk surface the level sets close into separate loops around separate minima, and counting the loops counts the basins.

Exercise 5.1 (Mark every zero-gradient point on a surface). A risk surface over two coefficients is

$$J(\vec{\beta}) = (\beta_1^2 - 1)^2 + (\beta_2^2 - 1)^2.$$

Find every point where $\nabla J = \vec{0}$.

Solution

Solution 5.1. The cost is a sum of one term in each coefficient, so each partial derivative involves one coefficient:

$$\frac{\partial J}{\partial \beta_j} = 4\beta_j(\beta_j^2 - 1), \quad j = 1, 2.$$

Each is zero at $\beta_j \in \{-1, 0, 1\}$, and the gradient vanishes when both are. That gives $3 \times 3 = 9$ points, the same three values as in Example 4.2, once in each coordinate. Figure 5 marks those nine points on the contours^a of J .

¹⁵<https://morrison-lab.github.io/mds/vector-calculus.html#def-level-set>

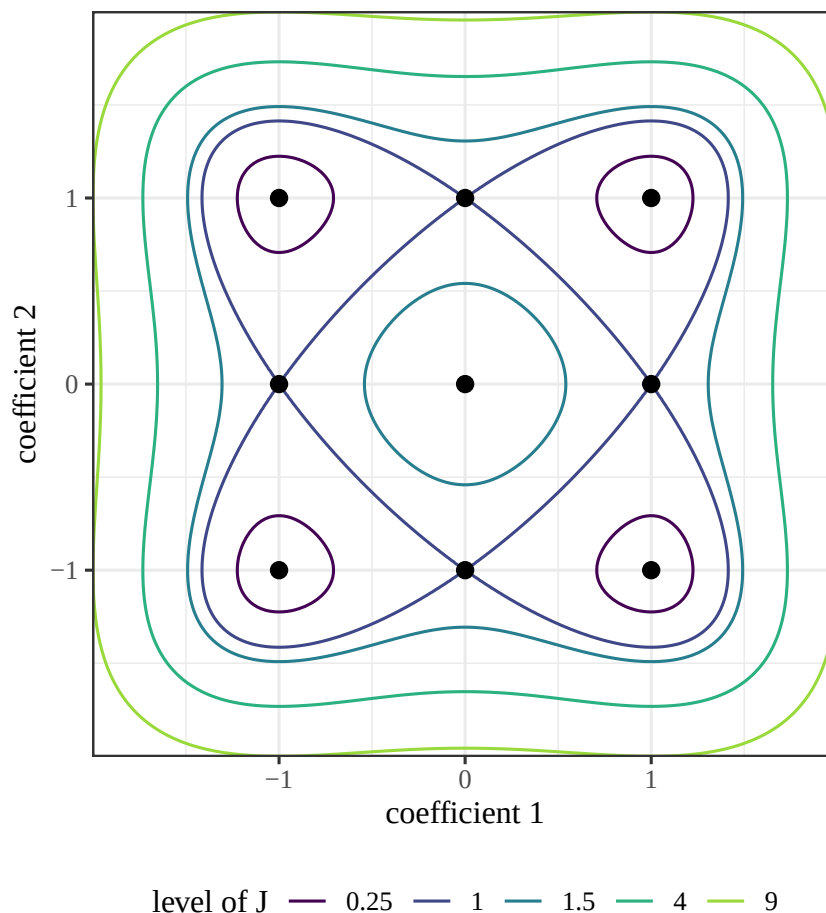


Figure 5: Contours^b of $J(\vec{\beta}) = (\beta_1^2 - 1)^2 + (\beta_2^2 - 1)^2$ at the levels 0.25, 1, 1.5, 4 and 9. The dots mark the nine points where $\nabla J = \vec{0}$.

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

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

Exercise 5.2 (Mark every zero-gradient point on a surface (part 2)). Use the setup of Exercise 5.1. Classify each point as a local minimizer^a, a local maximizer^b or a saddle point^c, using the sign of $\frac{\partial^2 J}{\partial \beta_1^2}$ and of $\frac{\partial^2 J}{\partial \beta_2^2}$. The mixed partial derivative is zero everywhere, so these two signs decide each case.

^a<https://morrison-lab.github.io/mds/algebra-basics.html#def-local-minimizer>

^b<https://morrison-lab.github.io/mds/optimization.html#def-local-maximizer>

^c<https://morrison-lab.github.io/mds/optimization.html#def-saddle-point>

Solution

Solution 5.2. The second derivatives are $\frac{\partial^2 J}{\partial \beta_j^2} = 12\beta_j^2 - 4$. It is $8 > 0$ where $\beta_j = \pm 1$ and $-4 < 0$ where $\beta_j = 0$. So the surface curves up along an axis where that coordinate is ± 1 and down where it is 0:

- Both coordinates ± 1 : curves up in both directions, a local minimum^a. There are four, $(\pm 1, \pm 1)$, each with $J = 0$.
- Both coordinates 0: curves down in both directions, a local maximum. It is the origin, with $J = 2$.
- One coordinate 0, the other ± 1 : curves up in one direction and down in the other, a saddle point^b.

There are four, $(0, \pm 1)$ and $(\pm 1, 0)$, each with $J = 1$.

```
# Every combination of the three stationary values for each coordinate.
points <- expand.grid(b1 = c(-1, 0, 1), b2 = c(-1, 0, 1))

# The second derivative along each axis, from the formula above.
points$curve1 <- 12 * points$b1^2 - 4
points$curve2 <- 12 * points$b2^2 - 4
points$J <- (points$b1^2 - 1)^2 + (points$b2^2 - 1)^2

# Both curvatures positive: minimum. Both negative: maximum. Mixed: saddle.
points$type <- ifelse(
  points$curve1 > 0 & points$curve2 > 0, "minimum",
  ifelse(points$curve1 < 0 & points$curve2 < 0, "maximum", "saddle")
)
points
```

```
# A tibble: 9 x 6
  b1    b2 curve1 curve2    J type
<dbl> <dbl> <dbl> <dbl> <dbl> <chr>
1   -1   -1      8      8     0 minimum
2    0   -1     -4      8     1 saddle
3    1   -1      8      8     0 minimum
4   -1    0      8     -4     1 saddle
5    0    0     -4     -4     2 maximum
6    1    0      8     -4     1 saddle
7   -1    1      8      8     0 minimum
8    0    1     -4      8     1 saddle
9    1    1      8      8     0 minimum
```

The table lists 4 minima, 1 maximum and 4 saddle points, matching the counts above.

^a<https://morrison-lab.github.io/mds/algebra-basics.html#def-local-minimizer>

^b<https://morrison-lab.github.io/mds/optimization.html#def-saddle-point>

Exercise 5.3 (Mark every zero-gradient point on a surface (part 3)). Use the setup of Exercise 5.1. Which of them can gradient descent^a settle at?

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

Solution

Solution 5.3. From almost every start, only at a local minimum^a, where every nearby point has a higher cost. Started exactly at the maximum the gradient is zero and descent never moves, and from every other start descent runs away from it. A saddle point^b attracts descent only from a thin set of starts. Along the line $\beta_1 = 0$ the cost is $1 + (\beta_2^2 - 1)^2$ in β_2 alone, so a start at $(0, 0.5)$ keeps $\beta_1 = 0$ and moves to the saddle point $(0, 1)$. In the same way, a start on $\beta_2 = 0$ with $\beta_1 \neq 0$, such as $(0.5, 0)$, keeps $\beta_2 = 0$ and converges to the saddle point $(\pm 1, 0)$. These lines have zero area, so a generic start, and any start with rounding error, leaves them and ends at a minimum (compare the single unstable start $\beta_0 = 0$ in Example 5.1, where the maximum has only that one start). Because the minima here are four, which one a run reaches depends on the start.

^a<https://morrison-lab.github.io/mds/algebra-basics.html#def-local-minimizer>

^b<https://morrison-lab.github.io/mds/optimization.html#def-saddle-point>

Exercise 5.4 (Sketch the level sets). For the surface of Exercise 5.1, sketch the level sets^a $\{\vec{\beta} : J(\vec{\beta}) = c\}$ for a value of c just above 0, for $c = 1$, and for a large c . Mark the nine zero-gradient points. How many

separate curves does each level set have?

^a<https://morrison-lab.github.io/mds/vector-calculus.html#def-level-set>

Solution

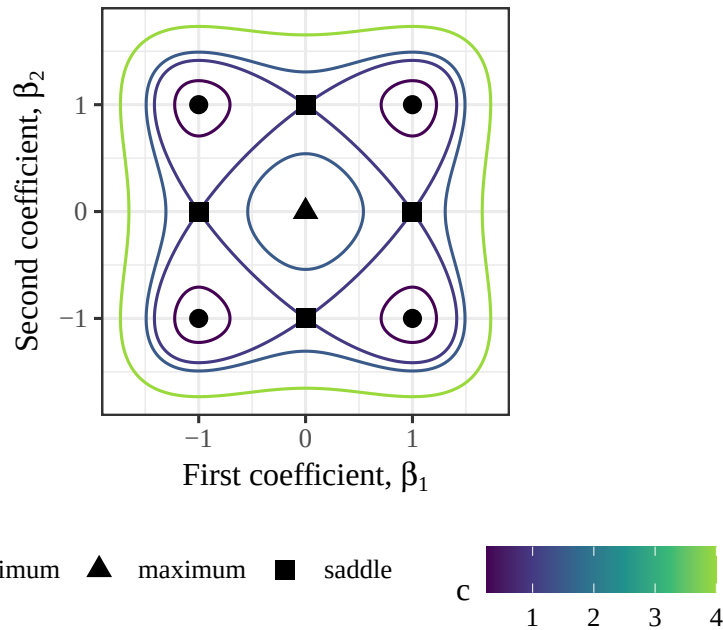
Solution 5.4. Near a local minimum the surface looks like a bowl, so its level sets there are small closed loops around the minimum. Near a saddle point the level set through the point is two curves that cross. The figure draws four values of c with the nine zero-gradient points marked.

5.3 R

```
library(ggplot2)

# The surface on a fine grid. The nine zero-gradient points, with their
# types, are the `points` table made in the previous solution.
grid <- expand.grid(
  b1 = seq(-2, 2, length.out = 300),
  b2 = seq(-2, 2, length.out = 300)
)
grid$J <- (grid$b1^2 - 1)^2 + (grid$b2^2 - 1)^2
points$type <- factor(points$type, levels = c("minimum", "maximum", "saddle"))

ggplot(grid) +
  aes(b1, b2) +
  geom_contour(aes(z = J, colour = after_stat(level)), breaks = c(0.25, 1, 1.5, 4)) +
  geom_point(aes(shape = type), data = points, size = 3) +
  scale_colour_viridis_c(name = "c", end = 0.85) +
  coord_fixed() +
  labs(
    x = expression("First coefficient, " * beta[1]),
    y = expression("Second coefficient, " * beta[2]),
    shape = NULL
  ) +
  theme(legend.position = "bottom")
```

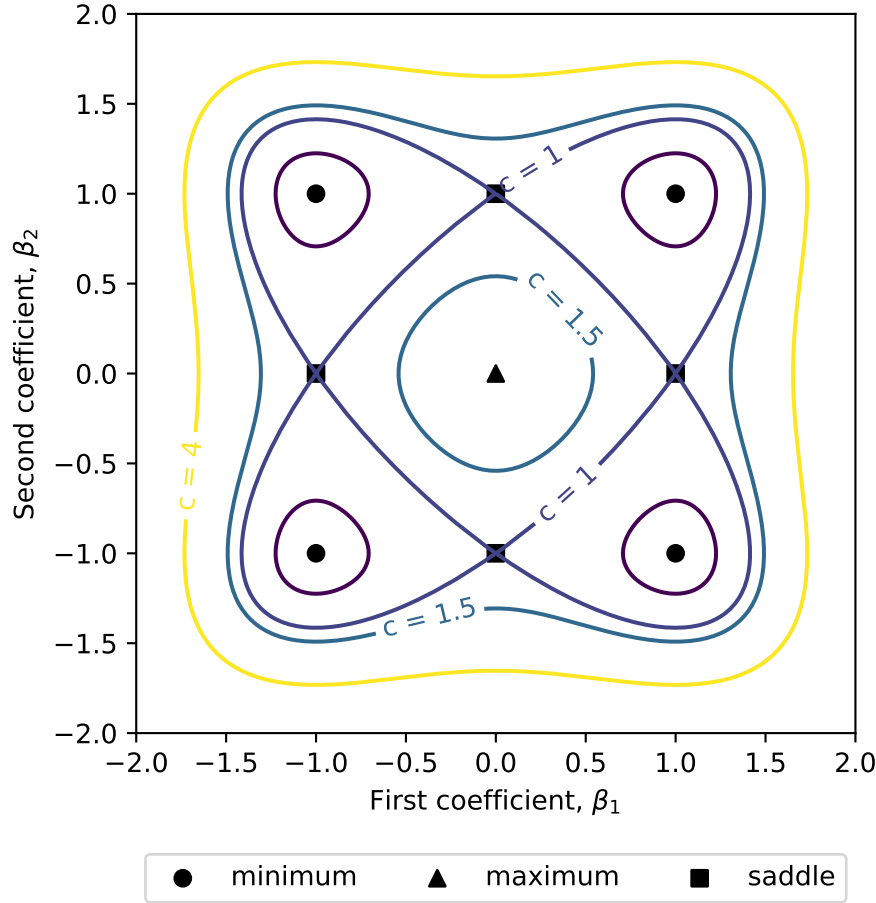


5.4 Python

```
import matplotlib.pyplot as plt
import numpy as np

# The surface on a fine grid.
b = np.linspace(-2, 2, 300)
b1, b2 = np.meshgrid(b, b)
J = (b1**2 - 1) ** 2 + (b2**2 - 1) ** 2

fig, ax = plt.subplots(figsize=(5, 5))
cs = ax.contour(b1, b2, J, levels=[0.25, 1, 1.5, 4], cmap="viridis")
ax.clabel(cs, fmt="c = %g");
# Minima at (+-1, +-1), the maximum at the origin, saddles on the axes.
ax.scatter([-1, -1, 1, 1], [-1, 1, -1, 1], marker="o", c="k", label="minimum");
ax.scatter([0], [0], marker="^", c="k", label="maximum");
ax.scatter([0, 0, -1, 1], [-1, 1, 0, 0], marker="s", c="k", label="saddle");
ax.set_aspect("equal");
ax.set_xlabel(r"First coefficient, $\beta_1$");
ax.set_ylabel(r"Second coefficient, $\beta_2$");
ax.legend(loc="upper center", bbox_to_anchor=(0.5, -0.15), ncol=3);
fig.tight_layout() # make room for the legend below the axes
plt.show()
```



For c just above 0 the level set is four small loops, one around each minimum. At $c = 1$ the four loops have grown until neighbouring ones touch, and they meet exactly at the four saddle points, so the level set is a single curve that crosses itself at each saddle. For $c = 1.5$, between the saddles' value 1 and the maximum's value 2, the loops have merged into an outer curve around all four minima and a small inner loop around the maximum. For a large c only the outer curve remains, and it is a single closed curve. Counting the loops at a small c counts the basins of attraction (Definition 5.1), and the saddles are where the basins touch.

5.5 Weight-space symmetry

A neural network's surface has a specific and easily described source of multiplicity. Take a hidden layer of L units and permute them — relabel unit 1 as unit 3 and so on — carrying each unit's incoming weights and its outgoing weights along with it. Every unit still receives the same inputs and still contributes the same quantity to the layer above, so the function the network computes does not change. There are $L!$ such relabelings, and a network with K hidden layers of L units each admits them independently in every layer, giving $(L!)^K$ weight vectors that compute one and the same function [9, Sec. 5.1.1].

If the hidden non-linearity is odd, as $\tanh$ is, a second family joins them. Negate every weight and the bias feeding one unit and that unit's output flips sign, because $\tanh(-a) = -\tanh(a)$; negate its outgoing weights as well and the flip cancels downstream. That is a further factor of 2^L per layer, so a $\tanh$ layer carries $L! 2^L$ symmetries rather than $L!$.

The consequence for optimization is that “this objective has an enormous number of local minima” overstates the difficulty. If one configuration is a local minimum then so is every one of its relabelings, and they are one solution written down in different orders. A surface with many equivalent copies of a minimum is no harder to descend than a surface carrying one copy.

5.6 Which obstacles actually matter

The original worry about non-convexity was local minima: with no convexity guarantee, nothing promises that the point found is the best one. That worry has largely not been confirmed in practice. The local minima that large networks actually reach tend to have training losses close to one another, and a local minimum bad enough to matter is hard to exhibit.

Saddle points and flat regions turned out to be the harder problem. A stationary point is a minimum only if the surface curves upward along *every* direction, and as the number of parameters grows that becomes an increasingly unlikely condition, so stationary points at a given height above the minimum are overwhelmingly saddles rather than minima [10]. Exactly at a saddle the gradient is zero and descent halts. A little to one side of it a descent direction exists, and the iterates eventually find it, so a saddle usually slows the run rather than ending it.

Definition 5.2 (Plateau). A **plateau** is a region where $\|\nabla J\|$ is small but nonzero, so the surface is nearly flat without containing a stationary point^a.

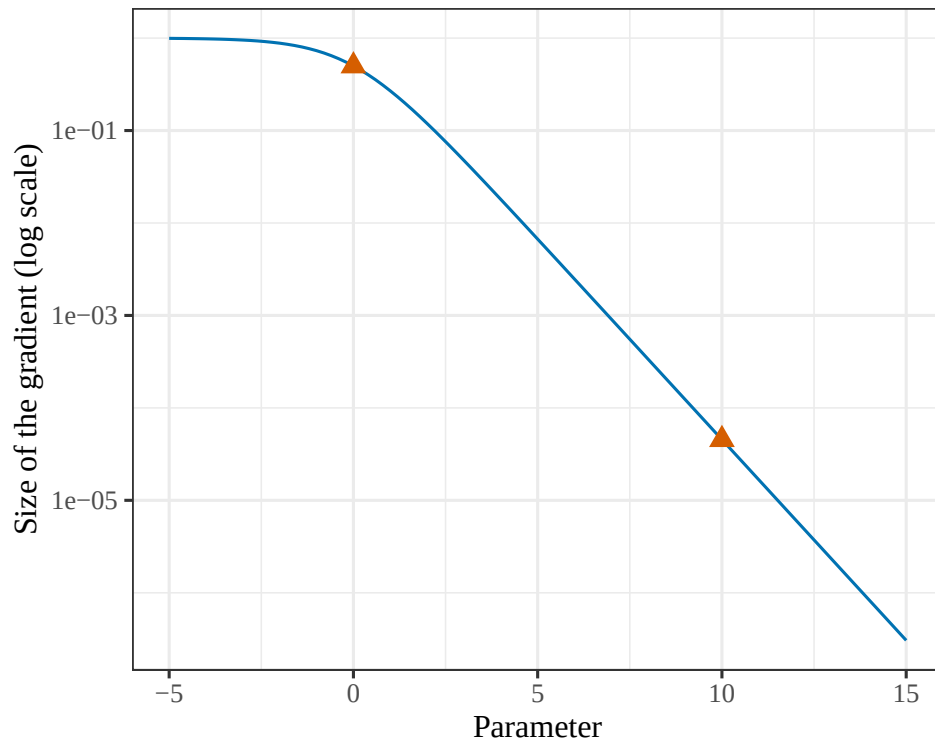
^a<https://morrison-lab.github.io/mds/optimization.html#def-stationary-point>

Example 5.2. Take the one-parameter logistic cost $J(\beta) = \log(1 + e^{-\beta})$, the loss of a single training point that the model should score as positive. Its derivative is $J'(\beta) = -1/(1 + e^{\beta})$, which is negative for every β , so J has no stationary point at all.

At $\beta = 0$ the gradient has size $|J'(0)| = 0.5$. At $\beta = 10$ it has size 4.54×10^{-5} , which is small but not zero. A descent step is the gradient times the step size, so the iterates keep moving to the right, more and more slowly. Everything to the right of about $\beta = 10$ is a plateau (Definition 5.2), and a run that enters it has not converged; it has only slowed down.

5.7 R

```
library(ggplot2)
pl_grad <- function(beta) 1 / (1 + exp(beta))
pl_marks <- data.frame(beta = c(0, 10))
pl_marks$size <- pl_grad(pl_marks$beta)
ggplot(data.frame(beta = c(-5, 15)), aes(beta)) +
  geom_function(fun = pl_grad, colour = "#0072B2") +
  geom_point(data = pl_marks, aes(beta, size), colour = "#D55E00", shape = 17, size = 3) +
  scale_y_log10() +
  labs(x = "Parameter", y = "Size of the gradient (log scale)")
```



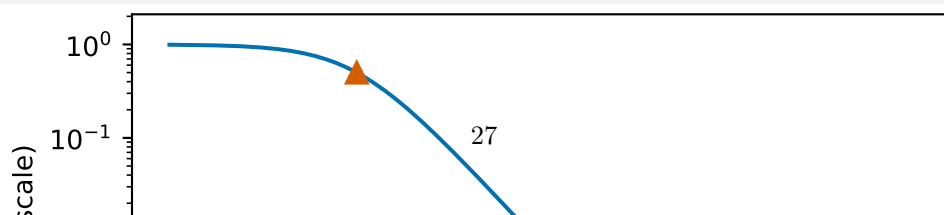
(a)

5.8 Python

```
import matplotlib.pyplot as plt
import numpy as np

def pl_grad(beta):
    return 1 / (1 + np.exp(beta))

grid_pl = np.linspace(-5, 15, 401)
marks_pl = np.array([0, 10])
fig, ax = plt.subplots(figsize=(5, 4), layout="constrained")
ax.plot(grid_pl, pl_grad(grid_pl), color="#0072B2");
ax.plot(marks_pl, pl_grad(marks_pl), color="#D55E00", marker="^", markersize=8, linestyle="");
ax.set_yscale("log");
ax.set_xlabel("Parameter");
ax.set_ylabel("Size of the gradient (log scale)");
plt.show()
```



On a plateau the gradient is small, so the steps are small, and progress can stall for a long stretch with nothing having gone wrong, in the sense of the algorithm being stuck. Saturated units are the usual cause: a unit sitting in the flat part of its non-linearity has a derivative near zero, so every weight feeding it receives almost no gradient. Backpropagation covers the extreme case, a unit whose derivative is exactly zero and stays there, and the two failures that depth adds on top of this one: vanishing and exploding gradients.

i Watch

Hutchinson covers critical points, saddle points and other optimization challenges^a (24 min) [5]. The video site requires a login, which enrolled students find in the syllabus.

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

6 Where to start

For a convex cost the starting point does not change the answer. Every run reaches the same minimum, so $\vec{\beta}_0$ decides only how long the run takes.

$\vec{\beta}_0 = \vec{0}$ is the usual choice, and it is not arbitrary. It is the null model: for logistic regression it predicts probability 1/2 for every observation, so the run begins from *no claim at all* and every subsequent step is evidence the data supplied. It is also the centre of an L_2 penalty, so a run stopped early has been shrunk toward the same point the penalty shrinks toward. That is why early stopping acts as a regularizer.

For a cost that is **not** convex the starting point decides which minimum you reach, and there is no longer one right answer to reach. Two things follow.

Starting at $\vec{\beta}_0 = \vec{0}$ becomes actively wrong for a neural network: every unit in a layer then computes the same function and receives the same gradient, so they stay identical forever and the layer has the capacity of a single unit. The weights have to be initialized randomly, to break that symmetry.

And the scale of that randomness matters separately. Draw the weights too large and the activations saturate, in exactly the sense of the flat regions this chapter keeps meeting; draw them too small and the signal shrinks toward zero through the layers. The standard schemes set the variance from the number of units feeding each layer, and the chapter on neural networks takes that up.

7 Choosing the step size

With a constant α there are two ways to go wrong, and the Week 6 notes name both:

- **Too big:** the iterates overshoot and $\|\vec{\beta}_t\| \rightarrow \infty$.
- **Too small:** every step is downhill, but there are far too many of them.

Figure 7 draws both failures, and a step size between them, on one elongated cost. Exercise 3.1 has already shown both ends of that range on real numbers. For least squares the boundary between them can be located exactly, because the cost is a quadratic and the whole trajectory can be written in closed form.

i Read

GeeksforGeeks explains the learning rate^a [11].

^a<https://www.geeksforgeeks.org/machine-learning/impact-of-learning-rate-on-a-model/>

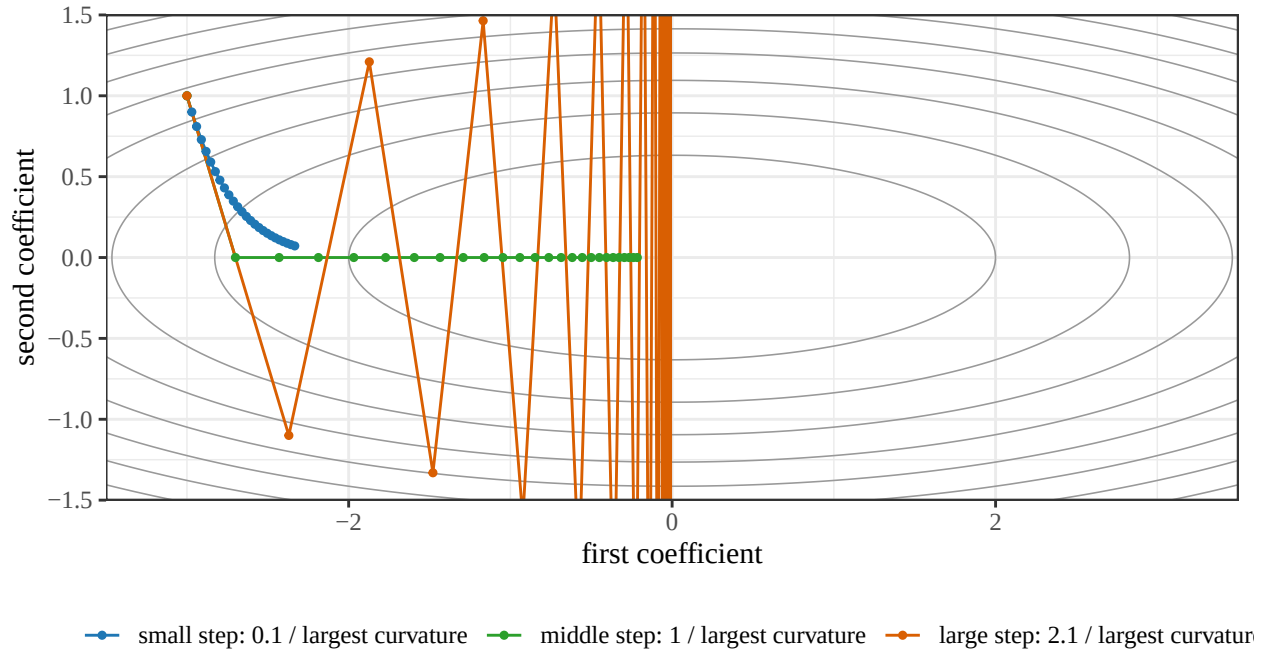


Figure 7: Gradient descent from one starting point on an elongated quadratic cost, with three constant step sizes, written as fractions or multiples of one over the cost’s largest curvature. The smallest step crawls along the valley. The middle step removes the error across the valley at once and then follows the valley floor. The largest step overshoots across the valley by more each time and moves away from the minimum. Grey curves are the cost’s level sets.

Exercise 7.1 (Rewrite the cost around its minimum). Before we can say which step sizes work, we need to know what gradient descent does to the *error* $e_t = \vec{\beta}_t - \vec{\beta}^*$ rather than to the cost. For least squares this can be worked out exactly, because the cost is a quadratic. Let $\vec{\beta}^*$ solve the normal equations. Show that

$$J(\vec{\beta}) = (\vec{\beta} - \vec{\beta}^*)^\top D^\top D(\vec{\beta} - \vec{\beta}^*) + J(\vec{\beta}^*)$$

by writing $D\vec{\beta} - y = D(\vec{\beta} - \vec{\beta}^*) + (D\vec{\beta}^* - y)$ and expanding.

Solution

Solution 7.1. Write $\tilde{r}^* = D\vec{\beta}^* - y$ for the residual at the minimum, and $\tilde{e} = \vec{\beta} - \vec{\beta}^*$. Then $D\vec{\beta} - y = D\tilde{e} + \tilde{r}^*$ and

$$\begin{aligned} J(\vec{\beta}) &= \|D\tilde{e} + \tilde{r}^*\|_2^2 \\ &= \tilde{e}^\top D^\top D\tilde{e} + 2\tilde{e}^\top D^\top \tilde{r}^* + \|\tilde{r}^*\|_2^2 \end{aligned}$$

The cross term vanishes. Solve the normal equations by hand showed $D^\top \tilde{r}^* = \vec{0}$ — the residual at the optimum is orthogonal^a to every column of D — so the middle term is $2\tilde{e}^\top \vec{0} = 0$, leaving

$$J(\vec{\beta}) = \tilde{e}^\top D^\top D\tilde{e} + J(\vec{\beta}^*)$$

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

Exercise 7.2 (Rewrite the cost around its minimum (part 2)). Use the setup of Exercise 7.1. Read off what the level sets^a of J look like, and what fixes their shape.

^a<https://morrison-lab.github.io/mds/vector-calculus.html#def-level-set>

Solution

Solution 7.2. J is a constant^a plus a quadratic form in $\tilde{e}$, so its level sets^b are ellipsoids centred on $\tilde{\beta}^*$. Their shape and orientation come entirely from $D^\top D$: the axes point along its eigenvectors, and an axis is long where the eigenvalue is small and short where it is large. The data determines the shape of the cost surface; y only determines where its centre is and what its minimum value is.

For our three points, $D^\top D = \begin{bmatrix} 3 & 3 \\ 3 & 5 \end{bmatrix}$, whose eigenvalues are $4 \pm \sqrt{10}$, so $\lambda_{\max} \approx 7.162$ and $\lambda_{\min} \approx 0.838$.

A level set^c of $\tilde{e}^\top D^\top D \tilde{e}$ has semi-axis length proportional to $1/\sqrt{\lambda}$, so it is the *square root* of the eigenvalue ratio that sets the shape: the contours are ellipses about $\sqrt{8.55} = 2.92$ times longer one way than the other. The long axis runs along the $\lambda_{\min}$ eigenvector, about 36° below the intercept^d axis, and the short axis along the $\lambda_{\max}$ eigenvector at right angles^e to it.

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

^b<https://morrison-lab.github.io/mds/vector-calculus.html#def-level-set>

^c<https://morrison-lab.github.io/mds/vector-calculus.html#def-level-set>

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

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

Exercise 7.3 (Rewrite the cost around its minimum (part 3)). Use the setup of Exercise 7.1. Use the identity to write ∇J in terms of $\tilde{e} = \tilde{\beta} - \tilde{\beta}^*$, and then show that one gradient-descent^a step gives

$$\tilde{e}_{t+1} = (I - 2\alpha D^\top D) \tilde{e}_t$$

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

Solution

Solution 7.3. Differentiating the identity, the constant^a drops out and

$$\nabla J(\tilde{\beta}) = 2D^\top D \tilde{e}$$

which is worth checking against the original form:

$$\begin{aligned} 2D^\top (D\tilde{\beta} - y) &= 2D^\top (D\tilde{e} + r^*) \\ &= 2D^\top D\tilde{e}, \end{aligned}$$

the same thing. Now subtract $\tilde{\beta}^*$ from both sides of the update:

$$\tilde{\beta}_{t+1} - \tilde{\beta}^* = \tilde{\beta}_t - \tilde{\beta}^* - \alpha \nabla J(\tilde{\beta}_t) = \tilde{e}_t - 2\alpha D^\top D \tilde{e}_t = (I - 2\alpha D^\top D) \tilde{e}_t$$

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

Exercise 7.4 (Rewrite the cost around its minimum (part 4)). Use the setup of Exercise 7.1. Explain why this one line, and no further algebra, already tells you that gradient descent^a on a quadratic either converges geometrically or diverges geometrically, with the boundary case in between held at constant^b size.

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

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

Solution

Solution 7.4. Because the recursion is *linear* and its matrix does not change from step to step. Iterating it gives $\tilde{e}_t = (I - 2\alpha D^\top D)^t \tilde{e}_0$, so the error is a fixed matrix raised to the power t . A matrix power either shrinks every component geometrically or blows one up geometrically, depending on that matrix's eigenvalues, and the cost follows the error by Solution 7.1: $J(\tilde{\beta}_t) - J(\tilde{\beta}^*)$ is a quadratic form in $\tilde{e}_t$, so it shrinks or grows like the square of whatever the error does.

There is no third behaviour to worry about, and no need to argue about it case by case. Exercise 7.6 turns this into a rule for α .

Exercise 7.5 (Find the exact range of step sizes that converge). Exercise 7.1 gave $\tilde{e}_{t+1} = (I - 2\alpha D^\top D)\tilde{e}_t$. Turn that into a condition on α .

Let $D^\top D = Q\Lambda Q^\top$ be its eigendecomposition^a, with eigenvalues $\lambda_1, \dots, \lambda_{p+1} \geq 0$. Write $\tilde{e}_t$ in the eigenvector basis and show that each coordinate evolves on its own, multiplied each step by a scalar you should name.

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

Solution

Solution 7.5. Since Q is orthogonal^a, $I - 2\alpha D^\top D = Q(I - 2\alpha\Lambda)Q^\top$. Let $\tilde{z}_t = Q^\top \tilde{e}_t$ hold the error's coordinates in the eigenvector basis. Then

$$\tilde{z}_{t+1} = (I - 2\alpha\Lambda)\tilde{z}_t, \quad \text{so} \quad z_{t+1,j} = (1 - 2\alpha\lambda_j)z_{t,j}$$

$I - 2\alpha\Lambda$ is diagonal, so the coordinates do not mix: each one is its own one-dimensional problem, multiplied every step by

$$\mu_j = 1 - 2\alpha\lambda_j$$

Gradient descent^b on a quadratic is $p + 1$ independent^c geometric sequences written as a matrix.

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

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

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

Exercise 7.6 (Find the exact range of step sizes that converge (part 2)). Use the setup of Exercise 7.5. State the condition on α under which every coordinate shrinks, and simplify it to a single upper bound.

Solution

Solution 7.6. Coordinate j shrinks exactly when $|\mu_j| < 1$, that is

$$-1 < 1 - 2\alpha\lambda_j < 1 \quad \Leftrightarrow \quad 0 < \alpha < \frac{1}{\lambda_j}$$

using $\lambda_j > 0$ for the division. Every coordinate shrinks when this holds for all j , and the binding case is the largest eigenvalue:

$$0 < \alpha < \frac{1}{\lambda_{\max}}$$

The *smallest* eigenvalue plays no part in whether we converge. It decides how *slowly*, which is Exercise 7.10. (If some $\lambda_j = 0$ then $\mu_j = 1$ and that coordinate never moves — the direction D cannot see; see When $D^\top D$ is not invertible. Gradient descent^a leaves it at its initial value, which is one way of choosing among the infinitely many solutions.)

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

Exercise 7.7 (Find the exact range of step sizes that converge (part 3)). Use the setup of Exercise 7.5. Evaluate that bound for our three-point D , where $\lambda_{\max} = 4 + \sqrt{10}$.

Solution

Solution 7.7.

$$\begin{aligned}\frac{1}{\lambda_{\max}} &= \frac{1}{4 + \sqrt{10}} \\ &= \frac{4 - \sqrt{10}}{16 - 10} \\ &= \frac{4 - \sqrt{10}}{6} \\ &\approx 0.1396\end{aligned}$$

Exercise 7.8 (Find the exact range of step sizes that converge (part 4)). Use the setup of Exercise 7.5. Compute the two multipliers for each of $\alpha = \frac{1}{20}$, $\alpha = \frac{1}{10}$, $\alpha = \frac{3}{20}$, and check the answers against what Exercise 3.1 actually observed.

Solution

Solution 7.8. With $\lambda_{\min} = 4 - \sqrt{10} \approx 0.8377$ and $\lambda_{\max} = 4 + \sqrt{10} \approx 7.1623$:

α	$\mu_{\min} = 1 - 2\alpha\lambda_{\min}$	$\mu_{\max} = 1 - 2\alpha\lambda_{\max}$	$\max_j \mu_j $	verdict
$1/20 = 0.05$	0.9162	0.2838	0.9162	converges
$1/10 = 0.10$	0.8325	-0.4325	0.8325	converges
$3/20 = 0.15$	0.7487	-1.1487	1.1487	diverges

This matches Exercise 3.1: the first two steps lowered the cost and the third raised it, and 0.15 is the only one of the three above the bound 0.1396 from Solution 7.7.

It also says something Exercise 3.1 could not. Judged after one step, $\alpha = 1/20$ looked best — it reached $J = 2.3$ where $\alpha = 1/10$ reached only 4.2. Judged over many steps, $\alpha = 1/10$ is the better choice, because its worst multiplier is 0.8325 against 0.9162. One step is not evidence about a step size.

Exercise 7.9 (Find the exact range of step sizes that converge (part 5)). Use the setup of Exercise 7.5. At $\alpha = 1/10$ one multiplier is negative. Say what that looks like if you plot the path, and whether it is a problem.

Solution

Solution 7.9. A negative multiplier means^a that coordinate of the error changes *sign* every step while shrinking in size, so the path zig-zags across the valley floor rather than sliding along it, crossing $\vec{\beta}^*$ in that direction on every step.

It is not a problem as long as $|\mu| < 1$. A step slightly too long for the steepest direction zig-zags like this while it is still short enough to converge, and here the zig-zag gives a better rate in the slow direction. It becomes a problem only at $|\mu| \geq 1$, where the zig-zag grows instead of shrinking — which is the $\alpha = 3/20$ row, and is exactly the picture of that step climbing the far wall in Exercise 3.1.

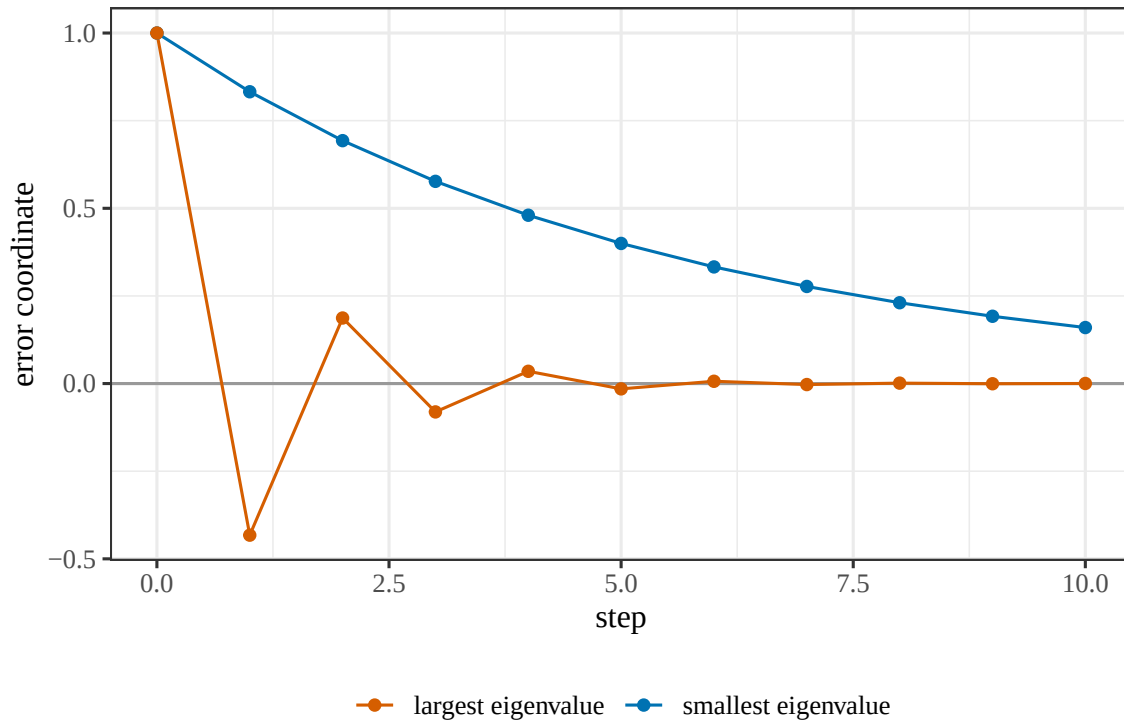


Figure 8: The two error coordinates of gradient descent^b at $\alpha = 1/10$. The coordinate with the largest eigenvalue has a negative multiplier, so it changes sign every step while it shrinks.

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

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

Exercise 7.10 (Find the best constant step size, and what limits it). Exercise 7.5 says which α converge. This one asks which converges *fastest*, and what limits that speed. The error shrinks each step by at worst $\rho(\alpha) = \max_j |1 - 2\alpha\lambda_j|$, so the best constant step size is the one minimizing ρ . Only the largest and smallest eigenvalues can attain the maximum. Sketch $|1 - 2\alpha\lambda_{\min}|$ and $|1 - 2\alpha\lambda_{\max}|$ as functions of α and argue that the best α is where the two curves cross.

Solution

Solution 7.10. Each $|1 - 2\alpha\lambda_j|$ is a V shape: it falls from 1 at $\alpha = 0$, reaches zero at $\alpha = 1/(2\lambda_j)$, and rises again. The V for $\lambda_{\max}$ turns first and climbs steeply; the V for $\lambda_{\min}$ is still falling gently. Every other eigenvalue sits between them, so the maximum of the whole family is the upper envelope of just these two.

For small α that envelope is the still-falling $|1 - 2\alpha\lambda_{\min}|$; for large α it is the rising $|1 - 2\alpha\lambda_{\max}|$. One is decreasing and the other increasing, so their maximum is minimized exactly where they meet.

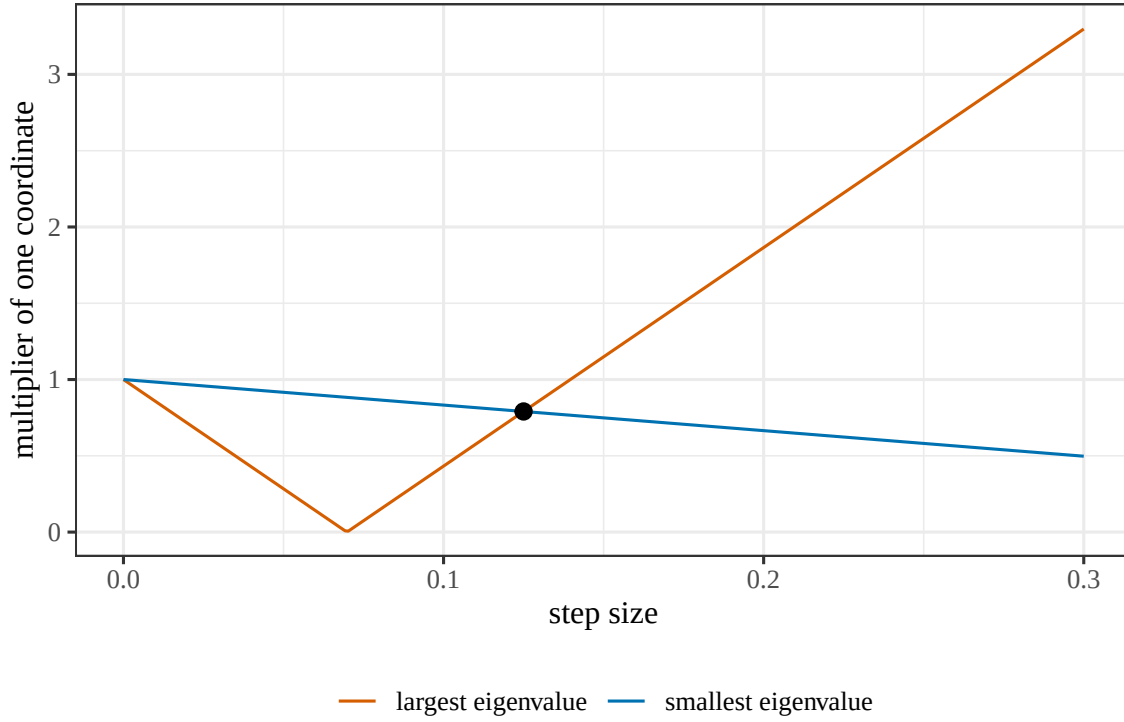


Figure 9: The multiplier $|1 - 2\alpha\lambda_j|$ of the coordinates with the smallest and the largest eigenvalue of the three-point design, against the step size α . The dot marks where the curves cross, at $\alpha = 1/(\lambda_{\min} + \lambda_{\max}) = 1/8$, the step size that minimizes the larger of the two.

Exercise 7.11 (Find the best constant step size, and what limits it (part 2)). Use the setup of Exercise 7.10.

Solve for that α , and show that the resulting rate is

$$\rho^* = \frac{\kappa - 1}{\kappa + 1}, \quad \kappa = \frac{\lambda_{\max}}{\lambda_{\min}}$$

the **condition number** of $D^\top D$.

Solution

Solution 7.11. At the crossing the $\lambda_{\max}$ branch is already past its turn, so $1 - 2\alpha\lambda_{\max}$ is negative and the equation is

$$1 - 2\alpha\lambda_{\min} = -(1 - 2\alpha\lambda_{\max}) \implies 2 = 2\alpha(\lambda_{\min} + \lambda_{\max}) \implies \alpha^* = \frac{1}{\lambda_{\min} + \lambda_{\max}}$$

Substituting back,

$$\begin{aligned} \rho^* &= 1 - \frac{2\lambda_{\min}}{\lambda_{\min} + \lambda_{\max}} \\ &= \frac{\lambda_{\max} - \lambda_{\min}}{\lambda_{\max} + \lambda_{\min}} \\ &= \frac{\kappa - 1}{\kappa + 1} \end{aligned}$$

dividing through by $\lambda_{\min}$ in the last step. Note $\alpha^* < 1/\lambda_{\max}$, so the best constant^a step is well inside the range Exercise 7.6 allows; the bound is not tight.

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

Exercise 7.12 (Find the best constant step size, and what limits it (part 3)). Use the setup of Exercise 7.10.

Evaluate α^* , κ and ρ^* for our three-point D . You may use $\lambda_{\min} + \lambda_{\max} = \text{tr}(D^\top D)$ to save arithmetic.

Solution

Solution 7.12. $\text{tr}(D^\top D) = 3 + 5 = 8$, so $\lambda_{\min} + \lambda_{\max} = 8$ and

$$\begin{aligned}\alpha^* &= \frac{1}{8} \\ &= 0.125\end{aligned}$$

with

$$\begin{aligned}\kappa &= \frac{4 + \sqrt{10}}{4 - \sqrt{10}} \\ &= \frac{(4 + \sqrt{10})^2}{6} \\ &= \frac{26 + 8\sqrt{10}}{6} \\ &= \frac{13 + 4\sqrt{10}}{3} \\ &\approx 8.5497\end{aligned}$$

$$\rho^* = \frac{\kappa - 1}{\kappa + 1} \approx 0.7906$$

Exercise 7.13 (Find the best constant step size, and what limits it (part 4)). Use the setup of Exercise 7.10.

How many steps does it take to cut the error by a factor of ten, at α^* and at each of $\alpha = 1/20$ and $\alpha = 1/10$?

Solution

Solution 7.13. Cutting the error by ten needs $\rho^t \leq 0.1$, so $t \geq \log(0.1)/\log \rho$.

α	ρ	steps per factor of ten
$1/8 = 0.125$	0.7906	$9.80 \rightarrow 10$
$1/10 = 0.1$	0.8325	$12.56 \rightarrow 13$
$1/20 = 0.05$	0.9162	$26.32 \rightarrow 27$

Halving the step from $1/10$ to $1/20$ roughly doubles the work, which is the “too small takes a long time” half of the constant-step warning made precise.

Exercise 7.14 (Find the best constant step size, and what limits it (part 5)). Use the setup of Exercise 7.10.

What happens to ρ^* as $\kappa \rightarrow \infty$?

Solution

Solution 7.14. As $\kappa \rightarrow \infty$, $\rho^* \rightarrow 1$: the rate approaches no progress at all, and the number of steps per digit of accuracy grows without bound. Gradient descent^a slows down in proportion to how badly conditioned the problem is, and no choice of constant^b step rescues it, because ρ^* is already the best available.

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

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

Exercise 7.15 (Find the best constant step size, and what limits it (part 6)). Use the setup of Exercise 7.10.

What does a large κ say about the data?

Solution

Solution 7.15. A large κ says the columns of D are close to collinear — there is a direction in feature space the data barely varies along, so the cost is nearly flat there and the contours are a long narrow valley. Descent alternates between the valley sides and advances slowly along its bottom.

Exercise 7.16 (Find the best constant step size, and what limits it (part 7)). Use the setup of Exercise 7.10.

Name one thing you could change to make κ smaller without changing the model.

Solution

Solution 7.16. The least expensive fix that leaves the model alone is to rescale the features, usually by centring each one and dividing by its standard deviation^a. A feature measured in millimetres and one measured in kilometres have columns a factor of 10^6 apart, and $D^\top D$ is quadratic in the column scale, so their eigenvalues come out about 10^{12} apart for reasons that have nothing to do with the data. Standardizing removes that entirely. The fitted function is unchanged — the coefficients simply absorb the scaling — but κ can fall by orders of magnitude, so the same algorithm^b converges in a fraction of the steps.

This is also the actual reason ridge regression often trains faster: adding λI to $D^\top D$ raises every eigenvalue by λ , which lifts $\lambda_{\min}$ proportionally far more than $\lambda_{\max}$ and so shrinks κ . That is a change to the model, though, not just to the arithmetic.

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

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

Theorem 7.1 (The best constant step size). *For gradient descent^a on the least squares cost with $\rho(h) = \max_j |1 - 2h\lambda_j|$, where $\lambda_1, \dots, \lambda_p$ are the eigenvalues of $D^\top D$, the rate is smallest at*

$$h^* = \frac{1}{\lambda_{\min} + \lambda_{\max}},$$

and then

$$\rho^* = \frac{\kappa - 1}{\kappa + 1}, \quad \kappa = \frac{\lambda_{\max}}{\lambda_{\min}}.$$

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

i Proof

Proof. By Solution 7.10 and Solution 7.11. □

Example 7.1 (A line fit on four points). Take the design matrix with a column of ones and the inputs 0, 1, 2, 3. Compute h^* and ρ^* from Theorem 7.1, and check them against a direct search over h :

7.1 Python

```
import numpy as np

X_ls = np.column_stack([np.ones(4), np.arange(4.0)])
ev = np.linalg.eigvalsh(X_ls.T @ X_ls)
h_star = 1 / (ev.min() + ev.max())
kappa = ev.max() / ev.min()
rho = lambda h: np.max(np.abs(1 - 2 * h * ev))
grid = np.linspace(1e-4, 0.2, 20001)
print(h_star, grid[np.argmin([rho(h) for h in grid])])

0.05555555555555555 0.055552260000000001

print((kappa - 1) / (kappa + 1), rho(h_star))

0.8678055195451838 0.8678055195451838
```

7.2 R

```
X_ls <- cbind(1, 0:3)
ev <- eigen(crossprod(X_ls))$values
h_star <- 1 / (min(ev) + max(ev))
kappa <- max(ev) / min(ev)
rho <- function(h) max(abs(1 - 2 * h * ev))
grid <- seq(1e-4, 0.2, length.out = 20001)
c(h_star, grid[which.min(sapply(grid, rho))])

[1] 0.05555556 0.05555226

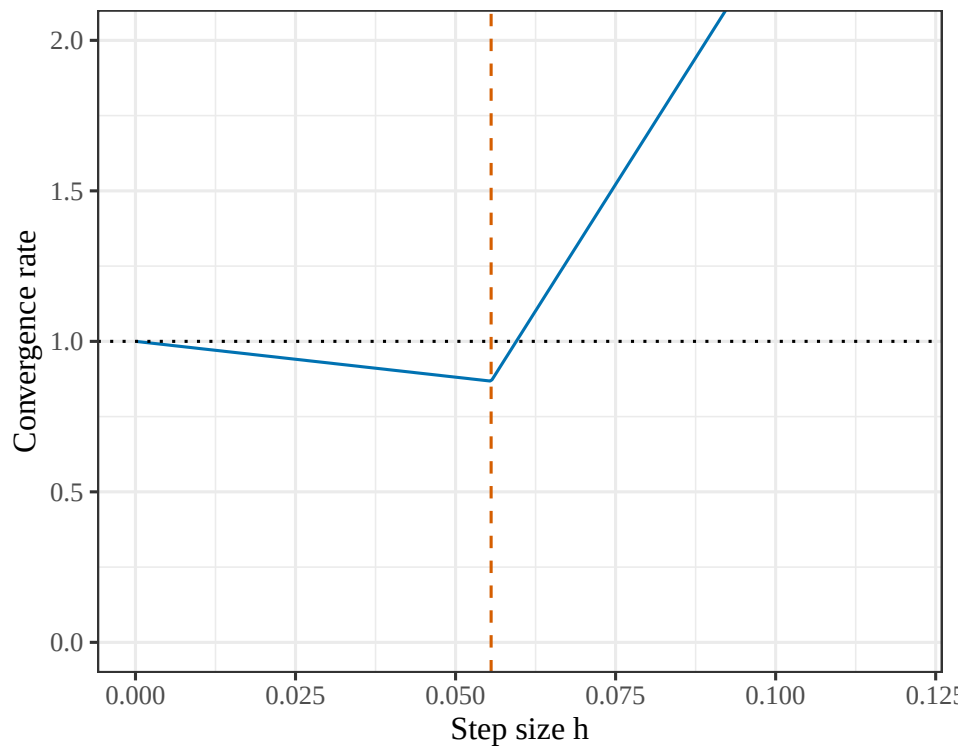
c((kappa - 1) / (kappa + 1), rho(h_star))

[1] 0.8678055 0.8678055
```

The search finds the same step size, and the rate matches the formula.

7.3 R

```
library(ggplot2)
oc <- data.frame(h = seq(1e-4, 0.12, length.out = 400))
oc$rate <- sapply(oc$h, rho)
ggplot(oc, aes(h, rate)) +
  geom_line(colour = "#0072B2") +
  geom_hline(yintercept = 1, linetype = "dotted") +
  geom_vline(xintercept = h_star, colour = "#D55E00", linetype = "dashed") +
  coord_cartesian(ylim = c(0, 2)) +
  labs(x = "Step size h", y = "Convergence rate")
```

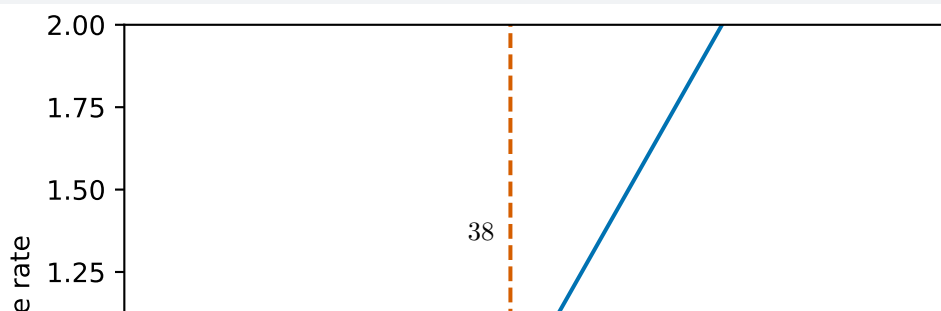


(a)

7.4 Python

```
import matplotlib.pyplot as plt
import numpy as np

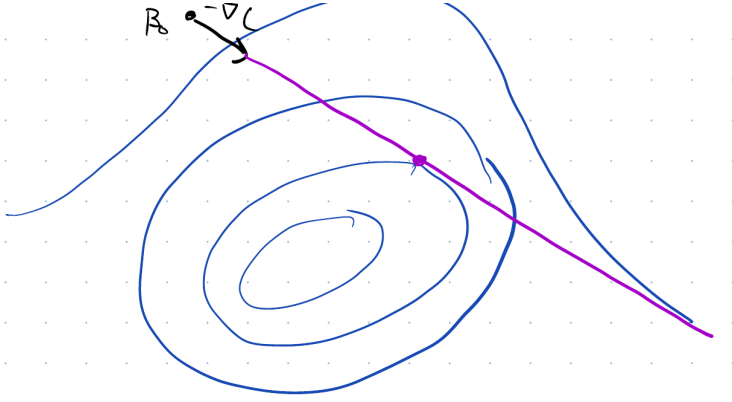
h_oc = np.linspace(1e-4, 0.12, 400)
fig, ax = plt.subplots(figsize=(5, 4), layout="constrained")
ax.plot(h_oc, [rho(h) for h in h_oc], color="#0072B2");
ax.axhline(1, color="black", linestyle=":");
ax.axvline(h_star, color="#D55E00", linestyle="--");
ax.set_ylim(0, 2);
ax.set_xlabel("Step size h");
ax.set_ylabel("Convergence rate");
plt.show()
```



7.5 Exact line search

Choosing α from Exercise 7.5 needed the eigenvalues of $D^\top D$, which we will not have for a general cost. The alternative is to stop choosing α in advance and choose it afresh at every step, by minimizing along the ray the gradient points down:

$$\alpha_t = \arg \min_{\alpha} C(\vec{\beta}_t - \alpha \nabla C(\vec{\beta}_t))$$



Exercise 7.17 (Choose the step size by minimizing along the ray). Instead of fixing α in advance, pick the best one at every step:

$$\alpha_t = \arg \min_{\alpha} C(\vec{\beta}_t - \alpha \nabla C(\vec{\beta}_t))$$

This is **exact line search**. It reduces a search over $\mathbb{R}^{p+1}$ to a search over a single variable, which for a quadratic can be done in closed form.

Write $\tilde{g} = \nabla J(\vec{\beta}_t)$ and $\tilde{r} = D\vec{\beta}_t - y$. Expand $J(\vec{\beta}_t - \alpha \tilde{g})$ as a quadratic in α .

Solution

Solution 7.17. $D(\vec{\beta}_t - \alpha \tilde{g}) - y = \tilde{r} - \alpha D\tilde{g}$, so

$$\begin{aligned} J(\vec{\beta}_t - \alpha \tilde{g}) &= \|\tilde{r} - \alpha D\tilde{g}\|_2^2 \\ &= \|\tilde{r}\|_2^2 - 2\alpha \tilde{r}^\top D\tilde{g} + \alpha^2 \|D\tilde{g}\|_2^2 \end{aligned}$$

a parabola in α , opening upward whenever $D\tilde{g} \neq \vec{0}$.

Exercise 7.18 (Choose the step size by minimizing along the ray (part 2)). Use the setup of Exercise 7.17. Minimize it over α , and use $\tilde{g} = 2D^\top \tilde{r}$ to simplify the answer to

$$\alpha_t = \frac{\|\tilde{r}\|_2^2}{2\|D\tilde{r}\|_2^2}$$

Solution

Solution 7.18. Differentiate and set to zero:

$$-2\tilde{r}^\top D\tilde{g} + 2\alpha \|D\tilde{g}\|_2^2 = 0 \implies \alpha_t = \frac{\tilde{r}^\top D\tilde{g}}{\|D\tilde{g}\|_2^2}$$

The numerator simplifies, because $\tilde{r}^\top D\tilde{g} = (D^\top \tilde{r})^\top \tilde{g}$ and $D^\top \tilde{r} = \frac{1}{2}\tilde{g}$:

$$\begin{aligned}\tilde{r}^\top D\tilde{g} &= \frac{1}{2}\tilde{g} \cdot \tilde{g} \\ &= \frac{1}{2}\|\tilde{g}\|_2^2\end{aligned}$$

giving $\alpha_t = \|g\|_2^2 / (2\|Dg\|_2^2)$. Both quantities are already computed on the way to the gradient, so the exact step costs one extra matrix–vector product.

Exercise 7.19 (Choose the step size by minimizing along the ray (part 3)). Use the setup of Exercise 7.17. Compute α_t at $\vec{\beta}_0 = (0, 0)^\top$ for our three points, in exact fractions.

Solution

Solution 7.19. At $\vec{\beta}_0 = \vec{0}$, Exercise 3.1 found $\tilde{g} = (-14, -18)^\top$ and $\tilde{r} = (-1, -3, -3)^\top$.

$$\begin{aligned}\|\tilde{g}\|_2^2 &= 196 + 324 \\ &= 520,\end{aligned}\quad D\tilde{g} = \begin{bmatrix} -14 \\ -32 \\ -50 \end{bmatrix}, \quad \begin{aligned}\|D\tilde{g}\|_2^2 &= 196 + 1024 + 2500 \\ &= 3720\end{aligned}$$

$$\begin{aligned}\alpha_0 &= \frac{520}{2 \cdot 3720} \\ &= \frac{520}{7440} \\ &= \frac{13}{186} \\ &\approx 0.0699\end{aligned}$$

Exercise 7.20 (Choose the step size by minimizing along the ray (part 4)). Use the setup of Exercise 7.17. Compute $\vec{\beta}_1$ and $J(\vec{\beta}_1)$, in exact fractions.

Solution

Solution 7.20.

$$\begin{aligned}\vec{\beta}_1 &= -\alpha_0 \tilde{g} \\ &= \frac{13}{186} (14, 18)^\top \\ &= \left(\frac{91}{93}, \frac{39}{31}\right)^\top \\ &\approx (0.978, 1.258)^\top\end{aligned}$$

$$\begin{aligned}J(\vec{\beta}_1) &= 19 - 2\alpha_0(260) + \alpha_0^2(3720) \\ &= \frac{77}{93} \\ &\approx 0.828\end{aligned}$$

Exercise 7.21 (Choose the step size by minimizing along the ray (part 5)). Use the setup of Exercise 7.17. Compare with the three constant^a steps of Exercise 3.1.

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

Solution

Solution 7.21. Against the constant^a steps, after a single step:

step rule	α	$J(\vec{\beta}_1)$
constant	$1/20 = 0.050$	2.30
exact line search	$13/186 \approx 0.070$	0.828
constant	$1/10 = 0.100$	4.20
constant	$3/20 = 0.150$	24.70

The minimum is $J(\vec{\beta}^*) = 2/3 \approx 0.667$, so one exact step has already removed most of the difference from the minimum. The chosen α lands between the two constant steps that worked — it is not simply “as large as possible”.

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

Exercise 7.22 (Choose the step size by minimizing along the ray (part 6)). Use the setup of Exercise 7.17. Show that consecutive exact-line-search directions are orthogonal^a: $\nabla J(\vec{\beta}_{t+1}) \cdot \nabla J(\vec{\beta}_t) = 0$.

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

Solution

Solution 7.22. α_t minimizes $\phi(\alpha) = J(\vec{\beta}_t - \alpha \tilde{g}_t)$, so $\phi'(\alpha_t) = 0$. By the chain rule,

$$\phi'(\alpha) = -\nabla J(\vec{\beta}_t - \alpha \tilde{g}_t) \cdot \tilde{g}_t$$

and at $\alpha = \alpha_t$ the first factor is $\nabla J(\vec{\beta}_{t+1})$, so $\nabla J(\vec{\beta}_{t+1}) \cdot \nabla J(\vec{\beta}_t) = 0$.

Exercise 7.23 (Choose the step size by minimizing along the ray (part 7)). Use the setup of Exercise 7.17. What does that say the path looks like in a long thin valley?

Solution

Solution 7.23. So every step is at right angles^a to the one before it, and the path has a staircase shape. In a long thin valley that is a disadvantage: the alternating right-angle turns make the iterates zig-zag across the valley while advancing very little along it, which is the same κ -dependent slowness as Exercise 7.10. Choosing the best step along each ray does not fix a badly conditioned problem; it just stops you overshooting while you progress slowly.

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

Exercise 7.24 (Choose the step size by minimizing along the ray (part 8)). Use the setup of Exercise 7.17. Give the reason exact line search is rarely used in practice, even though it is the best step along the ray.

Solution

Solution 7.24. Because the closed form in Solution 7.18 exists only for a quadratic. For anything else — logistic regression, a neural network — $\arg \min_{\alpha}$ is itself an optimization problem, solved approximately by repeatedly evaluating C , and each evaluation is a full pass over the data. Spending several passes to perfect one step is a wasteful use of computation when an approximate step costs one pass and gets most of the benefit.

Exercise 7.25 does this: it asks for *enough* decrease rather than the most.

7.6 Backtracking line search

Exact line search is too expensive away from the quadratic case, so in practice we settle for a step that is good enough rather than best. Start at $\alpha = 1$, evaluate the cost there, accept if the decrease is large enough, and otherwise shrink α and try again.

This is backtracking line search¹⁷, and “large enough” is the Armijo condition¹⁸ that Exercise 7.25 works through.

Line search and step-size selection are covered properly in chapter 9 of *Convex Optimization* by Stephen Boyd and Lieven Vandenbergh, which is freely readable at <https://web.stanford.edu/~boyd/cvxbook/>. Section 9.2, “Descent methods”, sets out both line searches used here; section 9.3, “Gradient descent method”, gives the convergence rate in terms of the condition number that Exercise 7.11 derives; and section 9.4, “Steepest descent method”, takes up the dependence on the choice of norm that Exercise 2.1 raises.

Exercise 7.25 (Backtracking: ask for enough decrease, not the most). Backtracking line search starts with a large step and shrinks it until the step is good enough. “Good enough” is the **Armijo condition**: for a fixed $c_1 \in (0, 1)$,

$$C(\vec{\beta}_t - \alpha \nabla C) \leq C(\vec{\beta}_t) - c_1 \alpha \|\nabla C\|_2^2$$

Start at $\alpha = 1$ and shrink by a fixed factor until it holds.

The right-hand side is the linear prediction of where the cost would go, discounted by c_1 . Say what the condition is comparing, and why $c_1 = 1$ would be the wrong thing to ask for.

Solution

Solution 7.25. It compares the decrease we actually got against the decrease the *linear model of the cost* predicted. Stepping α along $-\nabla C$ should, to first order, lower the cost by $\alpha \|\nabla C\|_2^2$ — that is the first-order term of Exercise 2.1 with the step length put back in. The condition accepts α when the real decrease is at least the fraction c_1 of the predicted one.

Asking for $c_1 = 1$ would demand the full linear prediction. A convex cost lies above its tangent line, so the real decrease is strictly less than the predicted decrease for every $\alpha > 0$, no step is ever accepted, and the line search keeps rejecting steps. It shrinks α toward zero and makes no progress. Our least-squares cost is convex, so this is what happens here.

A non-convex cost can curve the other way and beat its tangent line over some range of α , so $c_1 = 1$ would not reject *every* step there. It would still be a bad choice: acceptance would then depend on the local curvature rather than on the step being good, which is the opposite of what a line-search condition is for.

Exercise 7.26 (Backtracking: ask for enough decrease, not the most (part 2)). Use the setup of Exercise 7.25.

Run the search at $\vec{\beta}_0 = (0, 0)^\top$ on our three points with $c_1 = 10^{-4}$, starting at $\alpha = 1$ and shrinking by $1/10$. Use $J(\vec{\beta}_0) = 19$ and $\|\nabla J\|_2^2 = 520$, and note that $\alpha = 1$ gives $J = 3219$.

Solution

Solution 7.26. The required decrease is $c_1 \alpha \cdot 520 = 0.052\alpha$, so the test is $J(\vec{\beta}_1) \leq 19 - 0.052\alpha$.

trial	α	$J(\vec{\beta}_1)$	required $\leq$	verdict
1	1	3219	18.948	reject
2	1/10	4.20	18.995	accept

¹⁷<https://morrison-lab.github.io/mds/optimization.html#def-backtracking-line-search>

¹⁸<https://morrison-lab.github.io/mds/optimization.html#def-armijo-condition>

Exercise 7.27 (Backtracking: ask for enough decrease, not the most (part 3)). Use the setup of Exercise 7.25. Which α is accepted?

Solution

Solution 7.27. So $\alpha = 1/10$ is accepted. The first trial fails badly — starting from $\vec{\beta}_0 = (0, 0)$, a full step lands at $(14, 18)$ — and the second is accepted immediately. With c_1 this small the condition is doing little more than insisting the cost went down at all.

Exercise 7.28 (Backtracking: ask for enough decrease, not the most (part 4)). Use the setup of Exercise 7.25. Repeat with $c_1 = 1/2$. You will need $J = 14.172$ at $\alpha = 1/100$.

Solution

Solution 7.28. Now the required decrease is $0.5 \cdot 520 \cdot \alpha = 260\alpha$.

trial	α	$J(\vec{\beta}_1)$	required $\leq$	verdict
1	1	3219	-241	reject
2	1/10	4.20	-7	reject
3	1/100	14.172	16.4	accept

The first two fail for an interesting reason: the value they demand is below the smallest J can ever take, so no cost surface could have satisfied them. The strict c_1 has driven the accepted step down to about a seventh of the one exact line search chose.

Exercise 7.29 (Backtracking: ask for enough decrease, not the most (part 5)). Use the setup of Exercise 7.25. Show that with $c_1 = 1/2$ no step longer than about 0.0705 can ever be accepted from this starting point, whatever the shrink factor.

Solution

Solution 7.29. The cost can never fall below $J(\vec{\beta}^*) = 2/3$, so the largest decrease available from $J = 19$ is $19 - 2/3 = 55/3$. The Armijo condition^a with $c_1 = 1/2$ demands a decrease of 260α , which is achievable only if

$$260\alpha \leq \frac{55}{3} \implies \alpha \leq \frac{55}{780} = \frac{11}{156} \approx 0.0705$$

Any longer step is rejected no matter what the cost surface does, because the demand is impossible rather than merely unmet.

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

Exercise 7.30 (Backtracking: ask for enough decrease, not the most (part 6)). Use the setup of Exercise 7.25. Compare that number with the exact-line-search step of Exercise 7.17.

Solution

Solution 7.30. That bound is a quick one, obtained from the lower bound of J alone. The quadratic gives the exact answer, and it is simpler. Writing the decrease from Exercise 7.17 as $\alpha\|g\|_2^2 - \alpha^2\|Dg\|_2^2$, the Armijo condition^a reads

$$\alpha\|g\|_2^2 - \alpha^2\|Dg\|_2^2 \geq c_1\alpha\|g\|_2^2 \iff \alpha \leq 2(1 - c_1)\alpha^*$$

So on a quadratic, $c_1 = 1/2$ is exactly the demand that the step be no longer than the exact minimizer $\alpha^* = 13/186 \approx 0.0699$, and the floor derived in Solution 7.29 was within one percent of it. Smaller c_1 relaxes the ceiling proportionally: $c_1 = 10^{-4}$ allows almost twice α^* , so Solution ?? accepted $\alpha = 1/10$.

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

Exercise 7.31 (Backtracking: ask for enough decrease, not the most (part 7)). Use the setup of Exercise 7.25.

What does a small c_1 gain, and what does it risk?

Solution

Solution 7.31. A small c_1 allows long steps. The test is nearly “did the cost go down”, so backtracking stops at the first trial that works and makes fast progress per iteration.

The risk is that it accepts steps that make very little progress relative to their length, since a step that barely decreases the cost still passes when c_1 is tiny. In the worst case the iterates can progress slowly, making vanishing improvements while never triggering a rejection, and convergence guarantees are lost. The usual compromise is $c_1 = 10^{-4}$ with a shrink factor of $1/2$ rather than $1/10$, which keeps the steps long while shrinking gently enough not to overshoot a good step size on the way down.

i Watch

- Weinberger discusses how to choose the step size, from 23:20^a [2], Lecture 12.
- He then compares gradient descent with Newton’s method, from 35:10^b.

^a<https://www.youtube.com/watch?v=o6FfdP2uYh4&t=1400s>

^b<https://www.youtube.com/watch?v=o6FfdP2uYh4&t=2110s>

8 Knowing when to stop

Three quantities are worth watching as the iterations run:

- the cost;
- the parameters;
- the size of the gradient.

The usual rule watches the cost, relative to its own size.

Definition 8.1 (Relative change stopping criterion). The **relative change stopping criterion** with tolerance $\text{tol} > 0$ stops gradient descent^a at the first step t for which

$$\frac{|C(\vec{\beta}_{t+1}) - C(\vec{\beta}_t)|}{|C(\vec{\beta}_t)|} < \text{tol}.$$

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

Example 8.1 (Stopping a descent on $C(\beta) = \beta^2 + 1$). Run gradient descent^a from $\beta_0 = 2$ with $\alpha = 0.25$, and stop at tolerance $\text{tol} = 0.01$:

8.1 Python

```
cost_rc = lambda b: b**2 + 1
beta_rc, h_rc, tol_rc = 2.0, 0.25, 0.01
for t in range(100):
    beta_next = beta_rc - h_rc * 2 * beta_rc
    change = abs(cost_rc(beta_next) - cost_rc(beta_rc)) / abs(cost_rc(beta_rc))
    if change < tol_rc:
        break
    beta_rc = beta_next
print(t, change)
5 0.0029182879377431907
```

8.2 R

```
cost_rc <- function(b) b^2 + 1
beta_rc <- 2
h_rc <- 0.25
tol_rc <- 0.01
for (t_rc in 0:99) {
    beta_next <- beta_rc - h_rc * 2 * beta_rc
    change <- abs(cost_rc(beta_next) - cost_rc(beta_rc)) / abs(cost_rc(beta_rc))
    if (change < tol_rc) break
    beta_rc <- beta_next
}
c(t_rc, change)
```

[1] 5.000000000 0.002918288

The criterion stops at step $t = 5$, where the update to $\beta_{t+1} = 0.031$ changes the cost by a relative 0.0029. That parameter is still not the minimizer 0: a small relative change in cost does not show that the parameter has stopped moving.

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

Exercise 8.1 (Decide when to stop). There is no step at which gradient descent reports that it has converged, so we need a rule. The usual one watches the relative decrease in cost:

$$\frac{|C(\vec{\beta}_{t+1}) - C(\vec{\beta}_t)|}{|C(\vec{\beta}_t)|} < \text{tol}$$

Compute this quantity for the first step of Exercise 3.2 at $\alpha = 1/20$, and say whether a tolerance of 10^{-5} would have stopped there.

Solution

Solution 8.1. C went from 19 to $23/10 = 2.3$, so

$$\frac{|2.3 - 19|}{19} = \frac{16.7}{19} \approx 0.879$$

Nowhere near 10^{-5} , so the rule keeps going — as it should, since $J = 2.3$ is still more than three times the minimum $2/3$.

Exercise 8.2 (Decide when to stop (part 2)). Use the setup of Exercise 8.1. Why divide by $|C(\vec{\beta}_t)|$ at all?

Solution

Solution 8.2. To make the rule scale-free. An absolute threshold depends on the units of the response, which have nothing to do with whether the fit has converged. Dividing by $|C(\vec{\beta}_t)|$ removes the units, so the rule asks the question a person actually has in mind: *is the cost still moving by a meaningful fraction of itself?*

Exercise 8.3 (Decide when to stop (part 3)). Use the setup of Exercise 8.1. Give a specific change to the problem that leaves the fit identical but changes the *absolute* decrease.

Solution

Solution 8.3. Multiply every y_i by 1000 — change the units of the response from metres to millimetres — and the fitted line is the same line in new units, but every cost is a million times larger and so is every absolute decrease. An absolute threshold would stop far too early in one unit system and far too late in the other, and nothing about the problem has changed.

Exercise 8.4 (Decide when to stop (part 4)). Use the setup of Exercise 8.1. A common choice is $\text{tol} \approx 1/\sqrt{n}$. Argue for it from what the cost is estimating, rather than from convention.

Solution

Solution 8.4. The cost is a sum, or average, over a finite sample of size n , and what we care about is the population^a quantity it estimates^b. That estimate^c carries sampling error of order $1/\sqrt{n}$: draw a different sample of the same size and the whole cost surface shifts by about that much, taking its minimizer with it.

So improvements in the training cost that are much smaller than $1/\sqrt{n}$ are not improvements in anything we can measure on new data. They are adjustments to the noise in this particular sample, which is overfitting^d. Setting $\text{tol} \approx 1/\sqrt{n}$ stops optimizing once the arithmetic gets finer than the data can justify.

This is a rough guide, not a theorem: the constant^e in front depends on the variance^f of the loss, and the 10^{-5} in the notes is the same idea with n around 10^{10} , or simply a number someone found worked.

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

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

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

^d<https://morrison-lab.github.io/rme/chapters/predictor-selection.html#def-overfitting>

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

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

Exercise 8.5 (Decide when to stop (part 5)). Use the setup of Exercise 8.1. Two other quantities could be watched instead: the size of the step $\|\vec{\beta}_{t+1} - \vec{\beta}_t\|$, and the size of the gradient $\|\nabla C(\vec{\beta}_t)\|$. Say what each one is good at and how each can mislead.

Solution

Solution 8.5. *Step size* $\|\vec{\beta}_{t+1} - \vec{\beta}_t\|$. Good when what you care about is the parameters themselves — a coefficient you intend to report and interpret. It misleads when the cost is flat in some direction: the

iterates can move a long way along the bottom of a valley while the fit barely changes, which reads as “still working” when the useful work has finished. For constant-step descent it is exactly $\alpha \|\nabla C(\vec{\beta}_t)\|$, so it inherits the properties of the gradient criterion as well, scaled by α .

Gradient size $\|\nabla C(\vec{\beta}_t)\|$. This measures distance to the condition we are actually solving — a vanishing gradient — rather than progress between two arbitrary iterates, and the cost criterion cannot do that at all: the cost keeps falling by less and less without ever telling you how far from stationary you are. It misleads in two ways: it is not scale-free either, and it cannot tell a minimum from a saddle point^a or a flat **plateau**, where the gradient is small and the cost is not near its lowest value.

In practice, check more than one. The relative-decrease rule alone cannot distinguish “converged” from “taking tiny steps”.

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

Exercise 8.6 (Decide when to stop (part 6)). Use the setup of Exercise 8.1.

The Week 6 notes add the warning “don’t use for neural networks (danger!)”. Explain what goes wrong there.

Solution

Solution 8.6. A neural network’s cost is not convex^a, and its descent path typically spends long stretches on **plateaus** where the cost moves very little and then falls sharply. A small relative decrease is therefore evidence of a **plateau**, not of a minimum, and stopping on it abandons training at a point whose only property is being flat.

Two more reasons in the same direction. Neural networks are trained with stochastic gradients (Exercise 10.1), so consecutive costs differ by sampling noise from the minibatch as much as by progress, and the ratio is dominated by that noise. And they are usually *meant* to be stopped early, on a validation metric rather than a training cost — the training cost will happily keep falling while the model gets worse on new data.

The standard practice is to stop on validation performance, or simply on a fixed budget of **epochs**, and to treat the training curve as a diagnostic.

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

9 The cost of a step

Everything so far has counted *iterations*. The total cost is the number of iterations times the cost of one. The rest of the chapter is motivated by that per-iteration cost.

Exercise 9.1 (Count what a step costs). Express the cost as a sum over the training points,

$$C(\vec{\beta}) = \sum_{i=1}^n \ell_i(\vec{\beta})$$

where ℓ_i is the loss on point i — squared error, logistic loss, or anything else.

How many arithmetic operations does one evaluation of ∇C take, in terms of n and p ?

Solution

Solution 9.1. $O(np)$.

The gradient of a sum is the sum of the gradients, so $\nabla C = \sum_{i=1}^n \nabla \ell_i$ — n terms, and each has to be computed and added.

Note this is $O(n)$ in the number of terms — the form the Week 6 notes give — with the p made explicit.

Exercise 9.2 (Count what a step costs (part 2)). Use the setup of Exercise 9.1. State any assumption you make about $\nabla \ell_i$.

Solution

Solution 9.2. The assumption is that one $\nabla \ell_i$ costs $O(p)$, which holds for every model in this course that puts $\vec{\beta}$ into the loss through the single inner product $\vec{\beta} \cdot \tilde{x}_i$: least squares, logistic regression, support vector machines. Forming that inner product is p multiplications, and the chain rule turns the scalar derivative into a vector by scaling $\tilde{x}_i$, another p .

Exercise 9.3 (Count what a step costs (part 3)). Use the setup of Exercise 9.1. Give the total cost of running T gradient-descent^a steps.

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

Solution

Solution 9.3. $O(npT)$, since every step is a fresh full evaluation. Nothing is reused between steps: $\vec{\beta}$ has changed, so every residual and every gradient has to be recomputed from scratch.

Exercise 9.4 (Count what a step costs (part 4)). Use the setup of Exercise 9.1. Compare the cost with solving the normal equations directly.

Solution

Solution 9.4. Solving the normal equations costs $O(np^2)$ to form $D^\top D$ plus $O(p^3)$ to factor it.

Exercise 9.5 (Count what a step costs (part 5)). Use the setup of Exercise 9.1. For which shapes of data is each faster?

Solution

Solution 9.5.

- **Small p , any n :** the direct solve wins outright. With $p = 10$ the cube is trivial, and the $O(np^2)$ pass is a single pass over the data where descent needs T of them.
- **Large p :** descent wins, because np^2 and p^3 both grow faster in p than npT does. With $p = 10^5$, p^3 is 10^{15} operations and $D^\top D$ is a 10^{10} -entry matrix that will not fit in memory, while a gradient step never forms it. Which of the two terms dominates the direct solve depends on n against p : the cube leads when $n \lesssim p$, and the $O(np^2)$ pass leads otherwise.
- **No closed form:** the comparison does not arise, and descent is the only option — which is the case for every model after this chapter.

The crossover is roughly T against p : iterate when you expect fewer steps than you have features.

Exercise 9.6 (Count what a step costs (part 6)). Use the setup of Exercise 9.1.

Suppose $n = 10^8$ and you want 1000 steps. How many passes over the data is that, and why does that number force a method that does not read all of it before each step?

Solution

Solution 9.6. One gradient evaluation touches all 10^8 points, so 1000 steps is 1000 passes over the data — 10^{11} gradient computations before the first answer.

That is the number that makes full-batch descent unusable at scale. The work is spent re-reading data that has not changed to compute a direction that changes only slightly, and the fix is to stop insisting on the exact gradient: sample it instead, which is [Stochastic gradient descent](#).

i Remark (How gradient descent relates to Newton's method)

Remark 9.1 (How gradient descent relates to Newton's method). Gradient descent steps along $-\nabla C(\vec{\beta}_{t-1})$, scaled by a step size α that you choose. Here ∇C is the gradient of the cost. Newton's method^a steps along $-\ell''(\vec{\beta}_{t-1})^{-1} \nabla C(\vec{\beta}_{t-1})$. It replaces the scalar α with the inverse^b of the Hessian^c of C , which sets both the direction and the length of the step. The Newton-Raphson algorithm^d in the statistics notes is the same method, applied to the gradient of the negative log-likelihood^e.

Newton's method needs no step size, and it reaches the minimizer of a quadratic cost in one step. Each step costs more, because it forms a $D \times D$ matrix and solves a linear system with it, where a gradient step only needs the gradient. Logistic regression works this comparison through on a fitted model.

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

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

^c<https://morrison-lab.github.io/mds/vector-calculus.html#def-hessian>

^d<https://morrison-lab.github.io/sds/intro-MLEs.html#finding-the-mle-using-the-newton-raphson-algorithm>

^e<https://morrison-lab.github.io/sds/intro-MLEs.html#def-loglik>

10 Stochastic gradient descent

A full gradient is an average over the whole training set. An average over a set of points is exactly the thing statistics has a standard answer for: estimate it from a sample of the points. [Stochastic gradient descent](#) takes that literally. At each step, it draws one training point at random and steps along *its* gradient, as though the training set consisted of that point alone.

Definition 10.1 (Stochastic gradient descent). **Stochastic gradient descent** (SGD) [12] is gradient descent^a that replaces the gradient of the cost by the gradient of one training point, chosen at random. Let the cost be the mean^b of the point losses, $C(\vec{\beta}) = \frac{1}{n} \sum_{i=1}^n \ell_i(\vec{\beta})$. At step t , draw an index $I_t \sim \text{Uniform}\{1, \dots, n\}$ and update:

$$\underline{d}_t = -\nabla \ell_{I_t}(\vec{\beta}_t), \quad \vec{\beta}_{t+1} = \vec{\beta}_t + \alpha_t \underline{d}_t.$$

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

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

Example 10.1 (The three possible stochastic steps). Use the three points of Exercise 3.1, with the squared-error loss $\ell_i(\vec{\beta}) = (\beta_0 + \beta_1 x_i - y_i)^2$. Its gradient is $\nabla \ell_i(\vec{\beta}) = 2(\beta_0 + \beta_1 x_i - y_i)(1, x_i)$. From $\vec{\beta}_0 = \vec{0}$ with $\alpha = 0.1$, compute the update for each possible draw of I_0 :

10.1 Python

```
import numpy as np

x_sgd = np.array([0.0, 1.0, 2.0])
y_sgd = np.array([1.0, 3.0, 3.0])
beta_sgd0 = np.zeros(2)
h_sgd = 0.1

def point_grad(beta, i):
    return 2 * (beta[0] + beta[1] * x_sgd[i] - y_sgd[i]) * np.array([1.0, x_sgd[i]])

for i in range(3):
    print(i + 1, beta_sgd0 - h_sgd * point_grad(beta_sgd0, i))

1 [0.2 0. ]
2 [0.6 0.6]
3 [0.6 1.2]
```

10.2 R

```
x_sgd <- c(0, 1, 2)
y_sgd <- c(1, 3, 3)
beta_sgd0 <- c(0, 0)
h_sgd <- 0.1

point_grad <- function(beta, i) {
  2 * (beta[1] + beta[2] * x_sgd[i] - y_sgd[i]) * c(1, x_sgd[i])
}

apply(1:3, function(i) beta_sgd0 - h_sgd * point_grad(beta_sgd0, i))

[,1] [,2] [,3]
[1,] 0.2 0.6 0.6
[2,] 0.0 0.6 1.2
```

The update depends on which point is drawn: the first point pulls only the intercept^a, and the second and third pull both coefficients. Each draw gives a different step.

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

The direction is wrong at every step and correct on average, and that is enough. This idea is older than machine learning. Herbert Robbins and Sutton Monro published it in 1951 as a way of finding the root of a function that can only be observed with noise [13]. Their conditions on the [step sizes](#) are still the ones in use.

Definition 10.2 (Robbins–Monro conditions). The **Robbins–Monro conditions** [13] on the [step sizes](#) $\alpha_1, \alpha_2, \dots$ of [stochastic gradient descent](#) are

$$\sum_{t=1}^{\infty} \alpha_t = \infty \quad \text{and} \quad \sum_{t=1}^{\infty} \alpha_t^2 < \infty.$$

The first lets the iterates travel any distance they need to. The second keeps the accumulated noise finite.

Example 10.2 (Which schedules meet the conditions). Compare partial sums of the fixed [step size](#) $\alpha_t = 0.1$ and the decaying step size $\alpha_t = 1/t$, up to n terms:

10.3 Python

```
import numpy as np

for n_terms in (10**2, 10**4, 10**6):
    t = np.arange(1, n_terms + 1)
    const = np.full(n_terms, 0.1)
    print(n_terms, const.sum(), (const**2).sum(), (1 / t).sum(), (1 / t**2).sum())
```

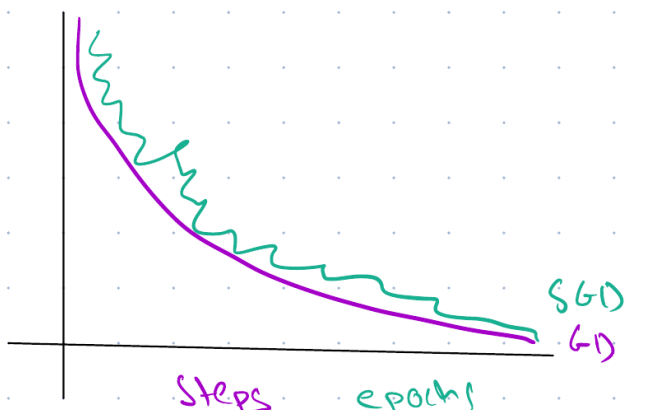
100 9.999999999999998 1.0000000000000004 5.187377517639621 1.6349839001848931
 10000 999.9999999999999 100.00000000000003 9.787606036044382 1.6448340718480599
 1000000 100000.00000000003 10000.000000000004 14.392726722865726 1.6449330668487263

10.4 R

```
sapply(c(1e2, 1e4, 1e6), function(n_terms) {
  t <- seq_len(n_terms)
  const <- rep(0.1, n_terms)
  c(n_terms = n_terms, const_sum = sum(const), const_sq = sum(const^2),
    inv_sum = sum(1 / t), inv_sq = sum(1 / t^2))
})
```

	[,1]	[,2]	[,3]
n_terms	100.000000	10000.000000	1.000000e+06
const_sum	10.000000	1000.000000	1.000000e+05
const_sq	1.000000	100.000000	1.000000e+04
inv_sum	5.187378	9.787606	1.439273e+01
inv_sq	1.634984	1.644834	1.644933e+00

The sums of 0.1 and of $1/t$ keep growing as n grows, so both meet the first condition. The sum of squares of $1/t$ levels off near $\pi^2/6 \approx 1.645$, so $1/t$ meets the second condition. The sum of squares of 0.1 keeps growing, so the fixed rate fails it.



Exercise 10.1 (Show that one random point points the right way on average). Express the cost as a sample mean^a rather than a sum:

$$C(\vec{\beta}) = \frac{1}{n} \sum_{i=1}^n \ell_i(\vec{\beta})$$

At step t , draw an index I_t uniformly from $\{1, \dots, n\}$ and step along that one point's gradient:

$$\vec{d}_t = -\nabla \ell_{I_t}(\vec{\beta}_t), \quad \vec{\beta}_{t+1} = \vec{\beta}_t + \alpha_t \vec{d}_t$$

Compute $\mathbb{E}[\vec{d}_t]$, showing the step where uniformity is used.

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

Solution

Solution 10.1.

$$\begin{aligned}\mathbb{E}[\tilde{d}_t] &= \sum_{i=1}^n \Pr[I_t = i] (-\nabla \ell_i(\vec{\beta}_t)) \\ &= \sum_{i=1}^n \frac{1}{n} (-\nabla \ell_i(\vec{\beta}_t)) \\ &= -\frac{1}{n} \sum_{i=1}^n \nabla \ell_i(\vec{\beta}_t) \\ &= -\nabla C(\vec{\beta}_t)\end{aligned}$$

Uniformity enters at the second equality, where every $\Pr[I_t = i]$ becomes $1/n$. That makes the average of the per-point gradients come out as the gradient of the average. This is the key step: the $1/n$ in the sampling distribution is the same $1/n$ that is in the cost.

Exercise 10.2 (Show that one random point points the right way on average (part 2)). Use the setup of Exercise 10.1.

What does the answer allow us to conclude about a single step, and what does it not?

Solution

Solution 10.2. It supports a statement about the *direction*: the stochastic step is an unbiased estimate^a of the full gradient step, so in expectation^b the method moves the way gradient descent^c would. It does not support any statement about a single step. $\tilde{d}_t$ can point uphill — Exercise 10.6 builds a case where it must — and the cost can rise for many steps in a row. Unbiasedness says the errors average out over many draws, not that any one draw is good. The practical consequence is that the cost curve of stochastic descent is a noisy band trending downward rather than the monotone curve a full-batch run gives, and that the step size has to shrink over time for the iterates to settle rather than fluctuate around the minimum forever.

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

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

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

Exercise 10.3 (Show that one random point points the right way on average (part 3)). Use the setup of Exercise 10.1.

Rewrite the cost as the sum $C(\vec{\beta}) = \sum_i \ell_i(\vec{\beta})$ instead. With $\tilde{d}_t$ unchanged, what is $\mathbb{E}[\tilde{d}_t]$ now?

Solution

Solution 10.3. Nothing about $\tilde{d}_t$ has changed, so its expectation^a is still the *mean*^b of the per-point gradients — but the cost's gradient is now the *sum*:

$$\mathbb{E}[\tilde{d}_t] = -\frac{1}{n} \sum_{i=1}^n \nabla \ell_i = -\frac{1}{n} \nabla C(\vec{\beta}_t)$$

The direction is right but the magnitude is off by a factor of n : each step is n times shorter than it should be.

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

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

Exercise 10.4 (Show that one random point points the right way on average (part 4)). Use the setup of Exercise 10.1.

What must change for the method to behave the same way?

Solution

Solution 10.4. Either scale the estimator^a, $\tilde{d}_t = -n\nabla\ell_{I_t}$, or scale the step size, $\alpha_t \rightarrow n\alpha_t$.

The two conventions differ only by that constant^b, and choosing one and using it consistently matters more than which. Exercise 10.14 shows what happens when they get mixed.

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

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

Exercise 10.5 (Show that one random point points the right way on average (part 5)). Use the setup of Exercise 10.1.

State the cost of one stochastic step in the notation of Exercise 9.1, and compare it with a full step.

Solution

Solution 10.5. $O(p)$ — one point's gradient, independent^a of n . A full step is $O(np)$, so a stochastic step is n times cheaper, and in the time one full step takes we can make n stochastic ones.

For the $n = 10^8$ of Exercise 9.1, that is the difference between 1000 full passes and a single pass that has already taken 10^8 steps. The estimate^b of each direction is far worse; there are vastly more of them.

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

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

Exercise 10.6 (Build a step that certainly goes uphill). Take the three points again, with $\ell_i(\vec{\beta}) = (\vec{\beta} \cdot \tilde{x}_i - y_i)^2$ the loss at point i and $J(\vec{\beta}) = \frac{1}{2} \sum_{i=1}^3 \ell_i(\vec{\beta})$ the cost this chapter has been minimizing throughout.

Compute $\nabla\ell_1, \nabla\ell_2, \nabla\ell_3$ at $\vec{\beta}_0 = (0, 0)^\top$, and check them against $\nabla J(\vec{\beta}_0) = (-14, -18)^\top$ from Exercise 3.1.

Solution

Solution 10.6. $\nabla\ell_i = 2(\vec{\beta} \cdot \tilde{x}_i - y_i)\tilde{x}_i$, and at $\vec{\beta}_0 = \vec{0}$ the predictions are all zero, so the residuals are $-1, -3, -3$:

$$\nabla\ell_1 = 2(-1) \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} -2 \\ 0 \end{bmatrix}, \quad \nabla\ell_2 = 2(-3) \begin{bmatrix} 1 \\ 1 \end{bmatrix} = \begin{bmatrix} -6 \\ -6 \end{bmatrix}, \quad \nabla\ell_3 = 2(-3) \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} -6 \\ -12 \end{bmatrix}$$

Their sum is $(-14, -18)^\top$, which is exactly $\nabla J(\vec{\beta}_0)$. The full gradient is the sum of the per-point gradients, which is the fact Exercise 10.1 rests on.

Exercise 10.7 (Build a step that certainly goes uphill (part 2)). Use the setup of Exercise 10.6.

Now start at the minimizer $\vec{\beta}^* = (4/3, 1)^\top$. Compute the three per-point gradients there.

Solution

Solution 10.7. Solving the normal equations by hand gave residuals $y - D\vec{\beta}^* = (-1/3, 2/3, -1/3)$ at the optimum. The gradient $\nabla\ell_i = 2(\tilde{x}_i \cdot \vec{\beta} - y_i)\tilde{x}_i$ carries the opposite, *fitted minus observed* sign, so the quantity entering each per-point gradient is $(1/3, -2/3, 1/3)$:

$$\nabla \ell_1 = \frac{2}{3} \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad \nabla \ell_2 = -\frac{4}{3} \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \quad \nabla \ell_3 = \frac{2}{3} \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Exercise 10.8 (Build a step that certainly goes uphill (part 3)). Use the setup of Exercise 10.6. What is their sum?

Solution

Solution 10.8. Their sum is $(\frac{2}{3} - \frac{4}{3} + \frac{2}{3}, 0 - \frac{4}{3} + \frac{4}{3})^\top = (0, 0)^\top$.

Exercise 10.9 (Build a step that certainly goes uphill (part 4)). Use the setup of Exercise 10.6. Why did it have to be that?

Solution

Solution 10.9. It had to be zero because $\vec{\beta}^\star$ minimizes J , so $\nabla J(\vec{\beta}^\star) = \vec{0}$, and the full gradient is that sum. Note what this does *not* say: no individual $\nabla \ell_i$ is zero. The minimum is where the per-point gradients cancel, not where they vanish. That cancellation is the whole source of the noise in stochastic descent — the individual terms stay large forever, and only their average dies.

Exercise 10.10 (Build a step that certainly goes uphill (part 5)). Use the setup of Exercise 10.6. Starting from $\vec{\beta}^\star$, take one stochastic step on point 1 with $\alpha = 1/20$. Compute J before and after, exactly.

Solution

Solution 10.10.

$$\begin{aligned} \tilde{d} &= -\nabla \ell_1 \\ &= -\frac{2}{3}(1, 0)^\top, \end{aligned}$$

so

$$\begin{aligned} \vec{\beta}_1 &= \vec{\beta}^\star + \frac{1}{20} \cdot \left(-\frac{2}{3}\right)(1, 0)^\top \\ &= \left(\frac{4}{3} - \frac{1}{30}, 1\right)^\top \\ &= \left(\frac{13}{10}, 1\right)^\top \end{aligned}$$

Using the identity of Exercise 7.1 with $\tilde{e} = (-1/30, 0)^\top$:

$$\begin{aligned} J(\vec{\beta}_1) - J(\vec{\beta}^\star) &= \tilde{e}^\top D^\top D \tilde{e} \\ &= 3\left(\frac{1}{30}\right)^2 \\ &= \frac{1}{300} \end{aligned}$$

so J rises from $\frac{2}{3} = \frac{200}{300}$ to $\frac{201}{300} = \frac{67}{100}$. Checking directly: the residuals at $(13/10, 1)$ are $(3/10, -7/10, 3/10)$, whose squares sum to $67/100$.

Exercise 10.11 (Build a step that certainly goes uphill (part 6)). Use the setup of Exercise 10.6. Full-batch gradient descent^a, started at $\vec{\beta}^\star$, does something different. What, and why?

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

Solution

Solution 10.11. Nothing at all. $\nabla J(\vec{\beta}^*) = \vec{0}$ by Solution 10.9, so the full-batch update $\vec{\beta}_1 = \vec{\beta}^* - \alpha \nabla J(\vec{\beta}^*)$ leaves $\vec{\beta}^*$ exactly where it is. The minimizer is a fixed point of full-batch descent, and is not a fixed point of the stochastic version.

Exercise 10.12 (Build a step that certainly goes uphill (part 7)). Use the setup of Exercise 10.6. Generalize: at the minimizer, is there *any* index whose stochastic step lowers the cost?

Solution

Solution 10.12. No. At $\vec{\beta}^*$ every step strictly increases J , whichever index is drawn. The cost is a strictly convex^a quadratic with its unique minimum at $\vec{\beta}^*$, so *any* move away from that point raises J — and no $\nabla \ell_i$ is zero, so every draw produces a move.

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

Exercise 10.13 (Build a step that certainly goes uphill (part 8)). Use the setup of Exercise 10.6. Given that answer, how should the step size behave as training proceeds?

Solution

Solution 10.13. So stochastic gradient descent^a does not converge to $\vec{\beta}^*$ and stay there. With a constant^b α it reaches a neighbourhood and then fluctuates within it, at a radius set by α and by how much the per-point gradients disagree. The fix is to shrink the step as training proceeds, so the fluctuation shrinks with it — $\alpha_t \propto 1/t$ is the classical schedule, small enough eventually to settle and large enough for long enough to have got there.

Compare Solution 10.11: full-batch descent needs no such schedule, because its estimate^c of the direction gets *exactly* right at the answer while the stochastic one never does. Reading only part of the data at each step forces the step size to shrink.

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

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

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

i Read

GeeksforGeeks explains stochastic gradient descent^a [14].

^a<https://www.geeksforgeeks.org/machine-learning/ml-stochastic-gradient-descent-sgd/>

10.5 Regularization with stochastic descent

Adding a penalty to the cost is the subject of ridge regression, and it interacts with sampling in a way that is easy to get wrong. The loss term is a sum over data points and so can be sampled; the penalty is a function of $\vec{\beta}$ alone and so cannot. If the penalty is scaled wrongly, you get a model whose regularization is n times too strong, not the model you asked for.

Exercise 10.14 (Add a penalty without making the estimator biased). Regularized training minimizes a data term plus a penalty:

$$C(\vec{\beta}) = \frac{1}{n} \sum_{i=1}^n \ell_i(\vec{\beta}) + \lambda R(\vec{\beta})$$

The penalty R is a function of $\vec{\beta}$ alone — it does not depend on any data point — so it cannot be sampled the way the loss can.
Write ∇C .

Solution

Solution 10.14. The gradient is linear, so it splits:

$$\nabla C(\vec{\beta}) = \frac{1}{n} \sum_{i=1}^n \nabla \ell_i(\vec{\beta}) + \lambda \nabla R(\vec{\beta})$$

Exercise 10.15 (Add a penalty without making the estimator biased (part 2)). Use the setup of Exercise 10.14.

Look for a stochastic estimator^a of the form $\tilde{d}_t = -\nabla \ell_{I_t}(\vec{\beta}_t) - c \nabla R(\vec{\beta}_t)$ with I_t uniform and c a constant^b. Compute $\mathbb{E}[\tilde{d}_t]$.

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

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

Solution

Solution 10.15. The penalty term is not random — R does not involve I_t — so it passes straight through the expectation^a:

$$\begin{aligned} \mathbb{E}[\tilde{d}_t] &= -\mathbb{E}[\nabla \ell_{I_t}] - c \nabla R \\ &= -\frac{1}{n} \sum_{i=1}^n \nabla \ell_i - c \nabla R \end{aligned}$$

using Exercise 10.1 for the first term.

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

Exercise 10.16 (Add a penalty without making the estimator biased (part 3)). Use the setup of Exercise 10.14.

Find the c that makes it unbiased.

Solution

Solution 10.16. Comparing with $-\nabla C$ from Solution 10.14, the data halves already agree, and the penalty halves agree exactly when

$$\boxed{c = \lambda}$$

So the correct estimator^a is

$$\tilde{d}_t = -\nabla \ell_{I_t}(\vec{\beta}_t) - \lambda \nabla R(\vec{\beta}_t)$$

The penalty gradient is applied in full at every step, unscaled. That is the point worth remembering: the loss is sampled and the penalty is not, because only the loss is a sum over data.

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

Exercise 10.17 (Add a penalty without making the estimator biased (part 4)). Use the setup of Exercise 10.14.

The Week 6 notes write $c = n\lambda$. Compute the bias^a that choice produces, and say by what factor the penalty is over-weighted.

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

Solution

Solution 10.17. With $c = n\lambda$,

$$\mathbb{E}[\tilde{d}_t] = -\frac{1}{n} \sum_{i=1}^n \nabla \ell_i - n\lambda \nabla R = -\nabla C - (n-1)\lambda \nabla R$$

so the bias^a is $-(n-1)\lambda \nabla R$, and the penalty is over-weighted by a factor of n . The estimator^b is unbiased only when $n = 1$.

Equivalently, the method is doing unbiased stochastic descent, but on the wrong objective: it is minimizing $\frac{1}{n} \sum_i \ell_i + n\lambda R$, a problem whose regularization strength is n times what was asked for.

A correction to the source notes

The Week 6 notes give this step as $\tilde{d}_t = -\nabla \ell_{I_t} - n\lambda \nabla R$, under the mean-cost convention $C = \frac{1}{n} \sum_i \ell_i + \lambda R$. It is off by a factor of n , as Solution 10.17 shows, and the line after it in those notes also drops a minus sign from the loss term. The corrected form is $\tilde{d}_t = -\nabla \ell_{I_t} - \lambda \nabla R$.

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

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

Exercise 10.18 (Add a penalty without making the estimator biased (part 5)). Use the setup of Exercise 10.14.

Now repeat the whole exercise under the sum convention $C(\vec{\beta}) = \sum_{i=1}^n \ell_i(\vec{\beta}) + \lambda R(\vec{\beta})$.

Solution

Solution 10.18. Under the sum convention,

$$\nabla C = \sum_{i=1}^n \nabla \ell_i + \lambda \nabla R$$

The sampled loss gradient still has expectation^a $\frac{1}{n} \sum_i \nabla \ell_i$, which is n times too small against a sum, so the loss term needs the factor:

$$\tilde{d}_t = -n \nabla \ell_{I_t}(\vec{\beta}_t) - \lambda \nabla R(\vec{\beta}_t)$$

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

Exercise 10.19 (Add a penalty without making the estimator biased (part 6)). Use the setup of Exercise 10.14.

Which term picks up a factor of n , and which does not?

Solution

Solution 10.19. The penalty term is unscaled again, so the factor of n goes on the sampled loss term only.

Exercise 10.20 (Add a penalty without making the estimator biased (part 7)). Use the setup of Exercise 10.14. State the rule in one sentence.

Solution

Solution 10.20. The rule in one sentence: **the factor of n belongs to whichever term was sampled, never to the penalty.** Under the mean^a convention nothing needs scaling; under the sum convention the loss does. The check that catches the mistake is the one Solution 10.17 performs: take the expectation^b of the step over I_t and require it to equal $-\nabla C$. The penalty does not depend on I_t , so any scaling error in it survives that expectation untouched and shows up at once as a leftover multiple of ∇R .

What does *not* work is setting $n = 1$. The error is a spurious factor of n , and at $n = 1$ that factor is 1, so the correct and the incorrect update coincide and the check passes on both. A test the bug passes is worse than no test.

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

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

Exercise 10.21 (Add a penalty without making the estimator biased (part 8)). Use the setup of Exercise 10.14.

Take ridge, $R(\vec{\beta}) = \|\vec{\beta}\|_2^2$, with $n = 10^6$. Describe what a run using $c = n\lambda$ produces, and say why nothing in the program reports an error.

Solution

Solution 10.21. Ridge has $\nabla R = 2\vec{\beta}$, so the direction carries a term $-2n\lambda\vec{\beta}_t$ instead of $-2\lambda\vec{\beta}_t$. With $n = 10^6$ the shrinkage toward the origin is a million times too strong, and it overwhelms the single data point's gradient at every step. The iterates collapse toward $\vec{\beta} = \vec{0}$ and stay there: the run produces the null model, predicting a constant^a regardless of the inputs^b. Keeping only that term, the iteration is $\vec{\beta}_{t+1} = (1 - 2nh\lambda)\vec{\beta}_t$, so the magnitude contracts whenever $0 < 2nh\lambda < 2$: monotonically below 1, and with the sign alternating between 1 and 2. Overshooting the origin is therefore not yet divergence. The run diverges only past $2nh\lambda > 2$, with $2nh\lambda = 2$ the boundary at which the iterates flip sign forever at constant size.

Nothing reports an error because nothing is wrong *as a program*. Every quantity is finite, every array has the right shape, the loop runs to its iteration limit, and the reported training cost decreases — on the objective the code is actually minimizing. The only symptom is a model that underfits badly, which looks exactly like a model whose λ was chosen too large. A practitioner would reach for cross-validation, find that every λ on the grid underfits, and conclude the features are uninformative.

This is the characteristic danger of an estimator^c bug: it does not crash, it silently minimizes a different objective. The defence is the calculation in Solution 10.15 — take the expectation^d of your update and check it against the gradient of the objective you meant to minimize.

^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-estimator>

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

10.6 Minibatches

One point per step is the cheapest possible estimate of the gradient and the noisiest. Averaging a handful of points lies between the two extremes, and essentially every real implementation does it.

Definition 10.3 (Minibatch gradient descent). **Minibatch gradient descent** is [stochastic gradient descent](#) with b indices B_t drawn uniformly and with replacement at each step, and the step taken along

their average gradient:

$$\tilde{d}_t = -\frac{1}{b} \sum_{i \in B_t} \nabla \ell_i(\vec{\beta}_t).$$

The number b is the **batch size**. With $b = 1$ it is stochastic gradient descent, and with $b = n$ it is full-batch descent in expectation^a.

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

Exercise 10.22 (Weigh a minibatch against a single point). In practice neither extreme is used. Let $\tilde{d}_t$ be the direction of minibatch gradient descent (Definition 10.3), and keep the mean-cost convention of Definition 10.1.

Show that $\tilde{d}_t$ is unbiased for $-\nabla C$, and that its variance^a is $1/b$ times that of the single-point estimator^b.

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

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

Solution

Solution 10.22. Each index is drawn uniformly, so each term has expectation $-\nabla C$ by Exercise 10.1, and the expectation of an average of b such terms is the same $-\nabla C$. Averaging cannot change an expectation.

For the variance, the draws are independent, so variances add:

$$\begin{aligned} \text{Var}\left(\frac{1}{b} \sum_{i \in B_t} \nabla \ell_i\right) &= \frac{1}{b^2} \cdot b \cdot \text{Var}(\nabla \ell_I) \\ &= \frac{1}{b} \text{Var}(\nabla \ell_I) \end{aligned}$$

taking the variance coordinatewise. This is the ordinary $1/b$ of a sample mean, and it is exactly why b helps: the direction is no more correct on average, it is just less scattered.

(Sampling b *distinct* indices, without replacement, does slightly better: the variance picks up a finite-population correction of $(n-b)/(n-1)$, which is negligible once $b \ll n$. That correction is not why implementations shuffle the data and slice it into consecutive batches. They do that because shuffling guarantees every observation is used exactly once per epoch, and because a contiguous slice reads from memory in order.)

Exercise 10.23 (Weigh a minibatch against a single point (part 2)). Use the setup of Exercise 10.22. Compute the variance^a of the single-point estimator^b at $\vec{\beta}^*$ for our three points, coordinate by coordinate, using the per-point gradients of Exercise 10.7.

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

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

Solution

Solution 10.23. From Exercise 10.7, the per-point gradients at $\vec{\beta}^*$ are

$$\nabla \ell_1 = \left(\frac{2}{3}, 0\right)^\top, \quad \nabla \ell_2 = \left(-\frac{4}{3}, -\frac{4}{3}\right)^\top, \quad \nabla \ell_3 = \left(\frac{2}{3}, \frac{4}{3}\right)^\top$$

Their mean^a is $\vec{0}$, so each variance^b is just the mean of the squares:

$$\begin{aligned}\text{Var}(\cdot)_{\text{intercept}} &= \frac{1}{3} \left(\frac{4}{9} + \frac{16}{9} + \frac{4}{9} \right) & \text{Var}(\cdot)_{\text{slope}} &= \frac{1}{3} \left(0 + \frac{16}{9} + \frac{16}{9} \right) \\ &= \frac{8}{9}, & &= \frac{32}{27}\end{aligned}$$

Both are of order one, at a point where the true gradient is exactly zero. That is the noise floor the schedule of Exercise 10.13 has to shrink against.

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

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

Exercise 10.24 (Weigh a minibatch against a single point (part 3)). Use the setup of Exercise 10.22. One step costs $O(bp)$. For a fixed budget of gradient evaluations, is a large b or a small b the better use of the budget, judged on this variance^a argument alone?

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

Solution

Solution 10.24. Neither, by this argument. A budget of K gradient evaluations buys K/b steps, each with variance^a σ^2/b . Doubling b halves the variance per step and halves the number of steps, so the two effects cancel: by this argument the budget is spent equally well either way, and any advantage goes to small b , because more steps means^b more chances to react to the curvature of the cost, and a step is where progress actually happens.

So the variance argument does not justify minibatching at all. It is worth noticing that, rather than assuming the standard practice must follow from the statistics.

^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>

Exercise 10.25 (Weigh a minibatch against a single point (part 4)). Use the setup of Exercise 10.22. Minibatches are nonetheless what everyone uses, and b between 32 and 512 is typical. Give the reason, and say why it is not the reason in Exercise 10.24.

Solution

Solution 10.25. Hardware. Computing b gradients as one $b \times p$ matrix multiplication is far faster than computing them one at a time, because the arithmetic units are vectorized and the memory is fetched in contiguous blocks — a GPU running $b = 256$ points may take barely longer than it takes for one. The budget in Solution 10.24 was counted in gradient evaluations; the budget that matters is counted in seconds, and those two stop being proportional as soon as the hardware can do work in parallel.

So b is chosen as the largest value that still fully uses the hardware and fits in memory, and then the step size is tuned for it. The typical range of 32 to 512 is a fact about processors, not about estimators^a. A second, smaller reason: the gradient noise itself acts a little like a regularizer, helping the iterates escape sharp minima in non-convex problems. That argues for *not* making b too large, and it cuts the opposite way from the hardware. So very large batches often need their own adjustments to work at all.

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

i Watch

- Andrew Ng contrasts batch and stochastic gradient descent, from 42:34^a [1], Lecture 2.
- He then describes stochastic gradient descent, from 44:56^b.

^ahttps://www.youtube.com/watch?v=4b4MUYve_U8&t=2554s

^bhttps://www.youtube.com/watch?v=4b4MUYve_U8&t=2696s

i Watch

Starmer works through stochastic gradient descent, from 4:47^a [15].

^a<https://www.youtube.com/watch?v=vMh0zPT0tLI&t=287s>

i Watch

Hutchinson covers minibatch stochastic gradient descent^a (19 min) [5].

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

i Read

GeeksforGeeks explains mini-batch gradient descent^a [16].

^a<https://www.geeksforgeeks.org/machine-learning/ml-mini-batch-gradient-descent-with-python/>

10.7 Epochs

Mini-batches are built by shuffling the training set and cutting the shuffled order into consecutive blocks of b .

Definition 10.4 (Epoch). An **epoch** is one complete pass through the training set.

Example 10.3 (Updates per epoch on the Auto data). The Auto data [17] have 392 complete cars. Count the updates in one epoch for several batch sizes b , when the short last block is kept and when it is dropped:

10.8 Python

```
import math
import pandas as pd

n_ep = len(pd.read_csv("data/auto.csv").dropna())
for b in (n_ep, 128, 32, 1):
    kept = math.ceil(n_ep / b)      # the short last block is a smaller batch
    dropped = n_ep // b            # the short last block is dropped
    print(b, kept, dropped)
```

```
392 1 1
128 4 3
32 13 12
1 392 392
```

10.9 R

```
n_ep <- nrow(na.omit(read.csv("data/auto.csv"))))
b_ep <- c(n_ep, 128, 32, 1)
data.frame(
  batch_size = b_ep,
  kept = ceiling(n_ep / b_ep),      # the short last block is a smaller batch
  dropped = floor(n_ep / b_ep)      # the short last block is dropped
)

# A tibble: 4 x 3
```

	batch_size	kept	dropped
	<dbl>	<dbl>	<dbl>
1	392	1	1
2	128	4	3
3	32	13	12
4	1	392	392

With $b = n = 392$ (full-batch descent), an epoch is one update. With $b = 32$ it is 13 updates when the last block of 8 cars is kept, and 12 when it is dropped. With $b = 1$ it is 392 updates. A run reported as 50 epochs therefore means very different numbers of updates depending on b .

In mini-batch descent, an epoch is one exhaustion of the blocks, after which the set is shuffled again and cut afresh. When n is not a multiple of b the last block is short, and it is either kept as a smaller batch or dropped. Dropping it is common, because a batch of an unusual size makes the noise of that one step differ from all the others.

The difference between an epoch and an update matters as soon as $b < n$. Full-batch descent makes one update per epoch, so the two words describe the same event and only one of them is needed. Mini-batch descent makes $\lfloor n/b \rfloor$ updates per epoch when the short final block is dropped, and $\lceil n/b \rceil$ when it is kept; a training run is usually reported in epochs while the optimizer is counted in updates. One epoch is almost never enough: with a small b a model may need many updates across many epochs before the parameters are any good. How many epochs is itself a hyperparameter, and the usual way to set it is to watch a held-out score and stop when it stops improving.

Read

GeeksforGeeks explains epochs^a [18].

^a<https://www.geeksforgeeks.org/machine-learning/epoch-in-machine-learning/>

11 Learning rate schedules

A constant step size cannot converge precisely under mini-batch sampling: the gradient noise does not vanish as the optimum is approached, so the iterates end up fluctuating around it. Near a minimum and for small α , the long-run (stationary) variance of the iterates grows in proportion to α , so the radius of the fluctuation scales as $\sqrt{\alpha}$ (and the excess loss, which is quadratic in that radius, scales as α). Shrinking α as training proceeds is the standard remedy, and the argument for it is the same at both ends of a run. Early on the steps should be long, because the parameters are far from any good solution and the noise is small next to the distance left to travel. Late on they should be short, because the remaining signal is small next to the noise, and a long step then mostly moves in the wrong direction.

The Robbins–Monro conditions introduced in Section 10 ($\sum \alpha_t = \infty$ and $\sum \alpha_t^2 < \infty$) make this requirement precise. A constant α satisfies the first and fails the second, so the noise keeps the iterates in a fluctuation of fixed size; a sequence like $\alpha_t = \alpha_0/t$ satisfies both. Several of the five practical schedules below satisfy only one of the two conditions and are used anyway, since a training run of fixed length needs no asymptotic guarantee; the conditions explain why decay is needed at all rather than prescribing a single schedule.

Definition 11.1 (Piecewise constant schedule). Let:

- α_0 be the starting **learning rate**, and α_t the rate at update t ;
- $\gamma \in (0, 1)$ be the drop factor;
- $t_1 < t_2 < \dots$ be the update counts at which the rate drops.

A **piecewise constant** schedule multiplies the rate by γ at each drop point:

$$\alpha_t = \alpha_0 \gamma^{k(t)}, \quad k(t) = \#\{j : t_j \leq t\}.$$

It is the simplest schedule, and still among the most used. Both of the things it asks for, the drop factor and the drop points, can be read off the loss curve of an earlier run.

Example 11.1 (A ten-fold drop twice). Let:

- α_0 be the starting [learning rate](#);
- γ be the drop factor;
- τ be the total number of updates.

Drop the rate at a third and at two-thirds of the run:

Take $\alpha_0 = 0.1$, $\gamma = 0.1$ and $\tau = 90$. The rate at each of the update counts $t = 0, 29, 30, 59, 60, 90$ is:

11.1 Python

```
import numpy as np

drops_pc = np.array(r.drops_pc)
for t in r.t_pc:
    print(int(t), round(r.lr0_pc * r.decay_pc ** np.sum(drops_pc <= t), 6))

0 0.1
29 0.1
30 0.01
59 0.01
60 0.001
90 0.001
```

11.2 R

```
data.frame(t = t_pc, rate = sapply(t_pc, function(t) lr0_pc * decay_pc^sum(drops_pc <= t)))

# A tibble: 6 x 2
      t rate
<dbl> <dbl>
1     0 0.1
2    29 0.1
3    30 0.01
4    59 0.01
5    60 0.001
6    90 0.001
```

The rate stays fixed between drops and falls by the factor γ exactly at $t = 30$ and $t = 60$.

Definition 11.2 (Linear decay). Let:

- α_0 be the starting [learning rate](#), and α_t the rate at update t ;
- α_τ be the final rate;
- τ be the number of updates over which the rate falls.

Linear decay [19] interpolates the rate between α_0 and α_τ over τ updates, then holds it at α_τ :

$$\alpha_t = (1 - \zeta)\alpha_0 + \zeta\alpha_\tau \quad \text{for } t \leq \tau, \quad \zeta = \frac{t}{\tau}.$$

Three numbers have to be chosen, and τ is the inconvenient one, since it fixes the number of updates in advance.

Example 11.2 (Linear decay between two rates). Let:

- α_0 be the starting [learning rate](#);
- α_τ be the final rate;
- τ be the number of updates over which the rate falls.

Take $\alpha_0 = 0.1$, $\alpha_\tau = 0.001$ and $\tau = 90$. The rate at $t = 0, 45, 90, 120$ is:

11.3 Python

```
for t in r.t_ld:
    frac = min(t / r.tau_ld, 1)
    print(int(t), (1 - frac) * r.lr0_ld + frac * r.lrT_ld)
```

```
0 0.1
45 0.0505
90 0.001
120 0.001
```

11.4 R

```
data.frame(t = t_ld, rate = rate_ld)
```

```
# A tibble: 4 x 2
      t   rate
<dbl> <dbl>
1     0 0.1
2    45 0.0505
3    90 0.001
4   120 0.001
```

The rate falls by the same amount at each update, then stays at 0.001 after update τ .

Definition 11.3 (Exponential decay). Let:

- α_0 be the starting [learning rate](#), and α_t the rate at update t ;
- $\gamma \in (0, 1)$ be the decay factor.

Exponential decay multiplies the rate by γ at every update:

$$\alpha_t = \alpha_0 \gamma^t.$$

The rate falls by the same proportion each step rather than by the same amount. It never reaches zero, and γ has to be set with the intended number of updates in mind.

Example 11.3 (The same factor over two run lengths). Let:

- α_0 be the starting [learning rate](#);
- γ be the decay factor.

Compare the rate after a short run with the rate after a long one:

Take $\alpha_0 = 0.1$ and $\gamma = 0.99$, and compare the rates at $t = 1000$ and $t = 100000$:

11.5 Python

```
for t in r.t_ed:
    print(int(t), r.lr0_ed * r.decay_ed**t)
```

```
1000 4.317124741065786e-06
100000 0.0
```

11.6 R

```
data.frame(t = t_ed, rate = rate_ed)
```

```
# A tibble: 2 x 2
      t   rate
<dbl> <dbl>
1  1000 0.00000432
```

2 100000 0

After 1000 updates the rate is $4.3\text{e-}06$, and after 100000 it is 0 in floating-point arithmetic. One factor gives two very different schedules.

Definition 11.4 (Patience). The **patience** of a training procedure that watches a held-out score is the number of consecutive evaluations the score may go without improving before the procedure reacts.

Example 11.4 (Two patiences on the same scores). Let:

- n_{up} be the number of evaluations over which the held-out score improves, after which it stays flat;
- p_{short} and p_{long} be two values of the **patience**.

Take $n_{\text{up}} = 8$, $p_{\text{short}} = 1$ and $p_{\text{long}} = 5$. The procedure first reacts at evaluation 9 with the short patience, and at evaluation 13 with the long one. A short patience reacts to random variation in the score, and a long one waits through many evaluations that could have been used.

Definition 11.5 (Performance decay). Let:

- α be the **learning rate**;
- $\gamma \in (0, 1)$ be the decay factor;
- p be the **patience**.

Performance decay (also called **plateau decay**) watches a held-out score during training. When the score has not improved for p evaluations, it multiplies α by γ and continues. It works like early stopping, but it cuts the learning rate instead of stopping. Unlike the other schedules, it looks at the data, and it needs no fixed number of updates in advance.

Example 11.5 (Halving the rate after three flat evaluations). Let:

- α_0 be the starting **learning rate**;
- γ be the decay factor;
- p be the **patience**.

Let the held-out score improve for the first 8 of 15 evaluations and then stay flat. Take $\alpha_0 = 0.1$, $\gamma = 0.5$ and $p = 3$:

11.7 Python

```
import numpy as np

lr_pd, best_pd, wait_pd = r.lr0_pd, -np.inf, 0
rates_pd = []
for s in r.score_pd:
    if s > best_pd:
        best_pd, wait_pd = s, 0
    else:
        wait_pd += 1
        if wait_pd == r.patience_pd:
            lr_pd, wait_pd = lr_pd * r.decay_pd, 0
        rates_pd.append(lr_pd)
print(rates_pd)
```

[0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.1, 0.05, 0.05, 0.05, 0.025, 0.025]

11.8 R

```

lr_pd <- lr0_pd
best_pd <- -Inf
wait_pd <- 0
rates_pd <- numeric(length(score_pd))
for (i in seq_along(score_pd)) {
  if (score_pd[i] > best_pd) {
    best_pd <- score_pd[i]
    wait_pd <- 0
  } else {
    wait_pd <- wait_pd + 1
    if (wait_pd == patience_pd) {
      lr_pd <- lr_pd * decay_pd
      wait_pd <- 0
    }
  }
  rates_pd[i] <- lr_pd
}
rates_pd

```

```

[1] 0.100 0.100 0.100 0.100 0.100 0.100 0.100 0.100 0.100 0.100 0.100 0.050 0.050
[13] 0.050 0.025 0.025

```

The rate stays at 0.1 while the score improves. It is cut by the factor γ after the third flat evaluation, at evaluation 11, and again every third flat evaluation after that.

i Two senses of “plateau”

A **plateau** of the cost is a region of the *parameter space* where the surface is flat. Plateau decay (Definition 11.5) is something else: a plateau in the held-out *score* as training proceeds, which triggers a cut in the **step size**. The first is a property of J , and the second is a property of the run.

Definition 11.6 (Cosine decay). Let:

- α_0 be the starting **learning rate**, and α_t the rate at update t ;
- τ be the number of updates in the run.

Cosine decay [20] takes the rate from α_0 down to zero along a half period of a cosine over τ updates:

$$\alpha_t = \frac{\alpha_0}{2} \left(1 + \cos \frac{\pi t}{\tau} \right).$$

Its derivative vanishes at both ends and is largest in the middle. The run gets a long stretch of nearly full-length steps, a quick transition, and a long stretch of fine ones.

Example 11.6 (Cosine decay over a fixed run). Let:

- α_0 be the starting **learning rate**;
- τ be the number of updates in the run.

Take $\alpha_0 = 0.1$ and $\tau = 90$. The rate at $t = 0, 15, 45, 75, 90$ is:

11.9 Python

```

import numpy as np

for t in r.t_cd:
    print(int(t), r.lr0_cd / 2 * (1 + np.cos(np.pi * t / r.tau_cd)))

```

```

0 0.1
15 0.09330127018922195

```

```
45 0.05
75 0.006698729810778065
90 0.0
```

11.10 R

```
data.frame(t = t_cd, rate = rate_cd)
```

```
# A tibble: 5 x 2
```

```
      t      rate
<dbl> <dbl>
```

```
1      0 0.1
2     15 0.0933
3     45 0.05
4     75 0.00670
5     90 0
```

The rate is 0.1 at the start, half of that at the midpoint $t = 45$, and zero at $t = 90$.

i Remark (Restarting the cosine schedule)

Remark 11.1 (Restarting the cosine schedule). Restarting the schedule periodically is a common variant: jump α back to α_0 and decay again. The argument for it is that the jump can move the iterates out of one [basin of attraction](#) and into a better one.

Piecewise constant, exponential, linear and cosine decay, and performance decay each add hyperparameters to a procedure that had one, and none of them can be set from the training loss alone.

Figure 11 compares four of these schedules on one set of axes.

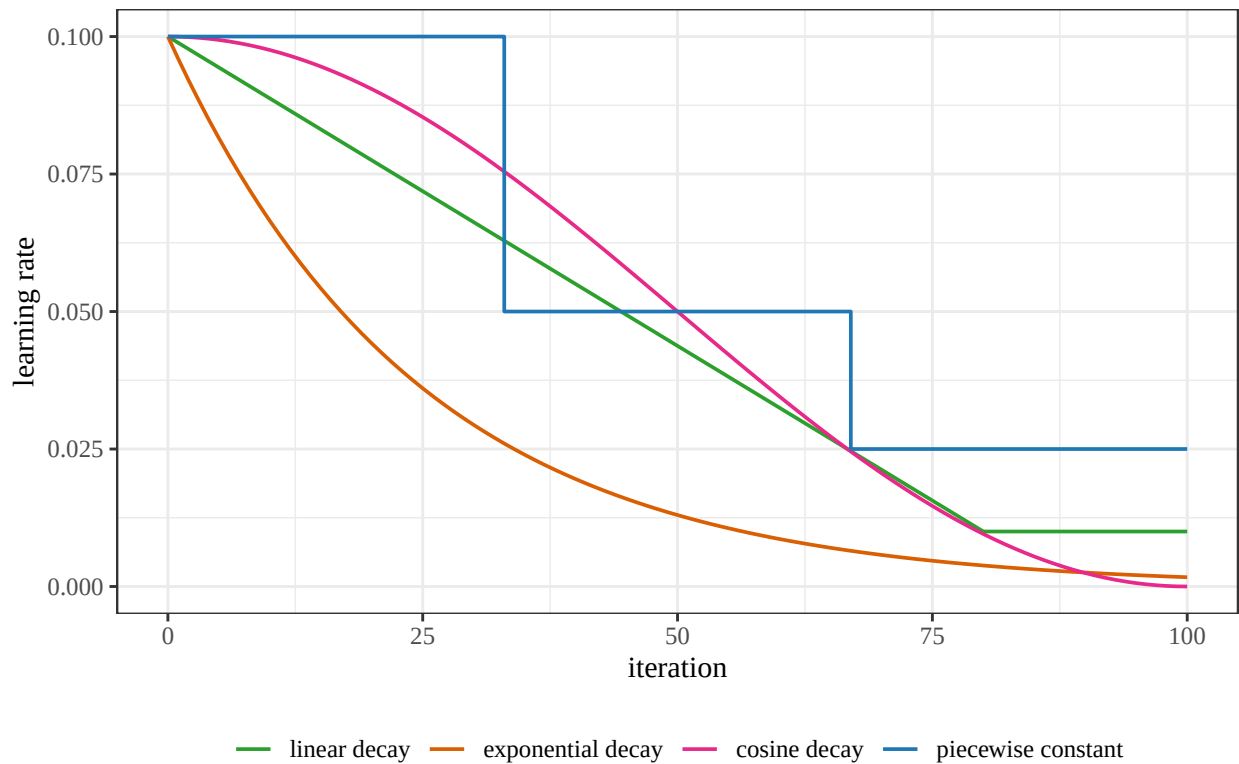


Figure 11: Four of the five [learning rate](#) schedules over 100 iterations starting from $\alpha_0 = 0.1$. [Performance decay](#) is left out because its drops depend on the validation score of an actual run. The [piecewise constant schedule](#) multiplies the rate by 0.5 at iterations 33 and 67. [Linear decay](#) interpolates to $\alpha_{80} = 0.01$ over 80 updates and then holds flat. [Exponential decay](#) multiplies by 0.96 at each step. [Cosine decay](#) follows a half cosine over $T = 100$ updates down to zero.

i Watch

The CS231N lecture on optimization discusses [learning rates](#) and their schedules, from 59:17^a [21], Lecture 3.

^a<https://www.youtube.com/watch?v=dyNGd06MWn4&t=3557s>

i Watch

Hutchinson covers learning rate schedules and adaptive methods^a (11 min) [5].

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

i Read

GeeksforGeeks explains learning rate decay^a [22].

^a<https://www.geeksforgeeks.org/machine-learning/learning-rate-decay/>

12 Momentum and adaptive step sizes

Plain gradient descent has two weaknesses that the earlier sections measured. On a cost whose contours are long, narrow ellipses, the steps zig-zag across the valley and creep along it, at a rate set by the condition

number (Exercise 7.11). And one step size α serves every coordinate, though the cost may be steep in some directions and nearly flat in others. Most optimizers used to train neural networks modify the update rule to address one or both.

The next exercise builds two of them and compares them with plain gradient descent on the simplest cost that shows the problem, $C(\vec{\beta}) = \frac{1}{2}(\theta_1^2 + 100\theta_2^2)$, whose condition number is 100.

Exercise 12.1 (Two ways to beat the zig-zag). Take $C(\vec{\beta}) = \frac{1}{2}(\theta_1^2 + 100\theta_2^2)$ with $\vec{\beta}_0 = (1, 1)$, so $\nabla C(\vec{\beta}) = (\theta_1, 100\theta_2)$. Count how many updates each method needs to bring the cost below 10^{-6} . Plain gradient descent^a with $\alpha = 0.019$. That is just under the largest step, $2/100$, that converges on this cost.

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

Solution

Solution 12.1.

12.1 Python

```
import numpy as np

# The cost 0.5 * (theta_1^2 + 100 * theta_2^2) and its gradient.
KAPPA = 100.0

def cost(theta):
    return 0.5 * (theta[0] ** 2 + KAPPA * theta[1] ** 2)

def grad(theta):
    return np.array([theta[0], KAPPA * theta[1]])

def updates_needed(step, state, goal=1e-6, cap=5000):
    # Run one update rule from (1, 1) until the cost drops below goal; the
    # cap stops a method that never gets there from looping forever.
    theta = np.array([1.0, 1.0])
    for t in range(1, cap + 1):
        theta, state = step(theta, state, t)
        if cost(theta) < goal:
            return t
    return None

def plain(theta, state, t, h=0.019):
    return theta - h * grad(theta), state

zero = np.zeros(2)
print("plain gradient descent:", updates_needed(plain, None))

plain gradient descent: 343
```

12.2 R

```

# The cost 0.5 * (theta_1^2 + 100 * theta_2^2) and its gradient.
kappa <- 100
cost <- function(theta) 0.5 * (theta[1]^2 + kappa * theta[2]^2)
grad <- function(theta) c(theta[1], kappa * theta[2])

updates_needed <- function(step, state, goal = 1e-6, cap = 5000) {
  # Run one update rule from (1, 1) until the cost drops below goal; the
  # cap stops a method that never gets there from looping forever.
  theta <- c(1, 1)
  for (t in seq_len(cap)) {
    out <- step(theta, state, t)
    theta <- out$theta
    state <- out$state
    if (cost(theta) < goal) {
      return(t)
    }
  }
  NA
}

plain <- function(theta, state, t, h = 0.019) {
  list(theta = theta - h * grad(theta), state = state)
}

cat("plain gradient descent:", updates_needed(plain, NULL), "\n")

plain gradient descent: 343

```

Definition 12.1 (Momentum coefficient). Let g_t be the gradient at update t , and $\underline{v}_t$ a running velocity built from them. The **momentum coefficient** $\beta \in [0, 1)$ is the weight on the previous velocity in

$$\underline{v}_t = \beta \underline{v}_{t-1} + g_t.$$

A gradient from k updates ago has weight β^k , so the velocity remembers roughly the last $1/(1 - \beta)$ updates.

Example 12.1 (How far back the velocity remembers). Let:

- β be the **momentum coefficient**;
- k be the number of updates since a gradient was added to the velocity.

For $\beta = 0, 0.5, 0.9, 0.99$, the weight on a gradient from $k = 10$ updates ago is 0, 0.00098, 0.35, 0.9, and the number of updates remembered, $1/(1 - \beta)$, is 1, 2, 10, 100 updates. At $\beta = 0$ the velocity is the current gradient alone, and at $\beta = 0.99$ it averages over about 100 updates.

Definition 12.2 (Momentum (heavy-ball method)). Let:

- C be the cost, and $\vec{\beta}_t$ the parameter vector after update t ;
- α be the **learning rate**;
- β be the **momentum coefficient**.

Momentum (the **heavy-ball method**) replaces the gradient in the update with a running velocity,

$$\underline{v}_t = \beta \underline{v}_{t-1} + \nabla C(\vec{\beta}_{t-1}),$$

and steps

$$\vec{\beta}_t = \vec{\beta}_{t-1} - \alpha \underline{v}_t.$$

At $\beta = 0$ this is plain gradient descent^a [23].

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

Example 12.2 (Momentum on a one-dimensional quadratic). Let:

- $C(\theta) = \frac{1}{2}\theta^2$ be the cost, so $\nabla C(\theta) = \theta$;
- θ_0 be the starting value of θ ;
- α be the **learning rate** and β the **momentum coefficient**;
- n be the number of updates.

Start at $\theta_0 = 1$ with $\alpha = 0.1$, and take $n = 20$ updates with $\beta = 0$ and with $\beta = 0.9$:

12.3 Python

```
def path_mo(beta):
    b, v, out = r.b0_mo, 0.0, []
    for _ in range(int(r.n_mo)):
        v = beta * v + b
        b = b - r.h_mo * v
        out.append(b)
    return out

plain, mom = path_mo(0.0), path_mo(r.beta_mo)
print(round(plain[-1], 4), round(mom[-1], 4), round(min(mom), 4))

0.1216 0.3447 -0.6037
```

12.4 R

```
round(c(plain = plain_mo[n_mo], momentum = mom_mo[n_mo], momentum_lowest = min(mom_mo)), 4)

      plain      momentum momentum_lowest
0.1216      0.3447      -0.6037
```

After $n = 20$ updates, plain descent has reached 0.122 and **momentum** 0.345. Momentum overshoots the minimizer 0, down to -0.604 at update 9, and then swings back. On a cost this simple, with a small **learning rate**, the swing makes momentum slower than plain descent. Its advantage shows on a long, narrow valley, as in the comparison below.

Exercise 12.2 (Two ways to beat the zig-zag (part 2)). Use the setup of Exercise 12.1.

Momentum: Keep a running velocity $\underline{v}_t = \beta \underline{v}_{t-1} + \nabla C(\vec{\beta}_{t-1})$ and step along it, $\vec{\beta}_t = \vec{\beta}_{t-1} - \alpha \underline{v}_t$, with $\underline{v}_0 = \vec{0}$, $\beta = 0.9$ and $\alpha = 0.01$.

Solution

Solution 12.2.

12.5 Python

```
def momentum(theta, v, t, h=0.01, beta=0.9):
    # The velocity accumulates past gradients, so steps along a steady
    # direction grow and steps that keep flipping sign cancel.
    v = beta * v + grad(theta)
    return theta - h * v, v

print("momentum: ", updates_needed(momentum, zero))
```

momentum: 138

12.6 R

```
momentum <- function(theta, v, t, h = 0.01, beta = 0.9) {
  # The velocity accumulates past gradients, so steps along a steady
  # direction grow and steps that keep flipping sign cancel.
  v <- beta * v + grad(theta)
  list(theta = theta - h * v, state = v)
}

cat("momentum: ", updates_needed(momentum, c(0, 0)), "\n")
```

momentum: 138

Momentum needs fewer updates than plain gradient descent^a on this cost. **Momentum** helps because the steep coordinate θ_2 keeps changing the sign of the gradient, so its velocity stays small, while the shallow coordinate θ_1 keeps the same sign, so its velocity builds up.

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

Definition 12.3 (Adaptive method). Let:

- $g_t = \nabla C(\vec{\beta}_{t-1})$ be the gradient at update t ;
- $\tilde{u}_t$ be a running average of the elementwise square g_t^2 ;
- $\epsilon > 0$ be a small positive number that avoids division by zero.

An **adaptive method** is a variant of gradient descent^a that gives each coordinate its own **learning rate**, set from the history of that coordinate's gradient. It divides each coordinate of the gradient by the root of the running average:

$$\vec{\beta}_t = \vec{\beta}_{t-1} - \alpha \frac{g_t}{\sqrt{\tilde{u}_t} + \epsilon}.$$

AdaGrad and RMSProp are early examples [12].

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

Example 12.3 (Two coordinates on very different scales). Let:

- g_t be the gradient, with two coordinates whose sizes are s_1 and s_2 at the first update;
- $\rho \in (0, 1)$ be the weight on the past in the running average, $\tilde{u}_t = \rho \tilde{u}_{t-1} + (1 - \rho) g_t^2$;
- q be the factor by which both coordinates of the gradient shrink at each update.

Take $s_1 = 100$, $s_2 = 0.01$, $\rho = 0.9$ and $q = 0.5$, and look at the first 5 updates. Dividing by the root of the running average gives the step $g_t / \sqrt{\tilde{u}_t}$ of each coordinate:

12.7 Python

```
import numpy as np

n = int(r.n_am)
g = np.outer(r.q_am ** np.arange(n), r.s_am)
u, steps = np.zeros(2), []
for t in range(n):
    u = r.rho_am * u + (1 - r.rho_am) * g[t] ** 2
    steps.append(g[t] / np.sqrt(u))
print(np.round(steps, 3))

[[3.162 3.162]
 [1.474 1.474]]
```

```
[0.755 0.755]
[0.395 0.395]
[0.208 0.208]]
```

12.8 R

```
round(step_am, 3)
```

```
      [,1] [,2]
[1,] 3.162 3.162
[2,] 1.474 1.474
[3,] 0.755 0.755
[4,] 0.395 0.395
[5,] 0.208 0.208
```

Plain gradient descent^a would move the first coordinate 10^4 times as far as the second. The scaled steps are the same for both coordinates, so the scale of a coordinate no longer matters. They also shrink from 3.16 to 0.21, because the running average still holds the earlier, larger values. Dividing by the current gradient alone would give a step of 1 at every update.

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

Definition 12.4 (Adam). **Adam** [24] is an **adaptive method** built from the gradient $\tilde{g}_t = \nabla C(\vec{\beta}_{t-1})$. It keeps exponential moving averages of the gradient and of its elementwise square:

$$\vec{\mu}_t = \beta_1 \vec{\mu}_{t-1} + (1 - \beta_1) \tilde{g}_t, \quad \tilde{u}_t = \beta_2 \tilde{u}_{t-1} + (1 - \beta_2) \tilde{g}_t^2.$$

It corrects each average for its start-up bias^a:

$$\hat{\vec{\mu}}_t = \frac{\vec{\mu}_t}{1 - \beta_1^t}, \quad \hat{\tilde{u}}_t = \frac{\tilde{u}_t}{1 - \beta_2^t}.$$

It then steps

$$\vec{\beta}_t = \vec{\beta}_{t-1} - \alpha \frac{\hat{\vec{\mu}}_t}{\sqrt{\hat{\tilde{u}}_t} + \epsilon}.$$

The defaults are $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$.

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

Exercise 12.3 (Two ways to beat the zig-zag (part 3)). Use the setup of Exercise 12.1.

Adam: Keep running averages of the gradient and of its elementwise square,

$$\vec{\mu}_t = \beta_1 \vec{\mu}_{t-1} + (1 - \beta_1) \tilde{g}_t, \quad \tilde{u}_t = \beta_2 \tilde{u}_{t-1} + (1 - \beta_2) \tilde{g}_t^2,$$

where $\tilde{g}_t = \nabla C(\vec{\beta}_{t-1})$, correct their start-up bias^a, $\hat{\vec{\mu}}_t = \vec{\mu}_t / (1 - \beta_1^t)$ and $\hat{\tilde{u}}_t = \tilde{u}_t / (1 - \beta_2^t)$, and step $\vec{\beta}_t = \vec{\beta}_{t-1} - \alpha \hat{\vec{\mu}}_t / (\sqrt{\hat{\tilde{u}}_t} + \epsilon)$, with the division and square root taken coordinate by coordinate. Use $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$ and $\alpha = 0.1$.

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

Solution

Solution 12.3.

12.9 Python

```
def adam(theta, state, t, h=0.1, b1=0.9, b2=0.999, eps=1e-8):
    m, u = state
    g = grad(theta)
    m = b1 * m + (1 - b1) * g          # running mean of the gradient
    u = b2 * u + (1 - b2) * g ** 2      # running mean of its square
    m_hat = m / (1 - b1 ** t)          # undo the bias from starting at zero
    u_hat = u / (1 - b2 ** t)
    # Dividing by the root of the second moment gives each coordinate its own step size.
    return theta - h * m_hat / (np.sqrt(u_hat) + eps), (m, u)

print("Adam:                ", updates_needed(adam, (zero, zero)))
```

Adam: 82

12.10 R

```
adam <- function(theta, state, t, h = 0.1, b1 = 0.9, b2 = 0.999, eps = 1e-8) {
  g <- grad(theta)
  m <- b1 * state$m + (1 - b1) * g # running mean of the gradient
  u <- b2 * state$u + (1 - b2) * g^2 # running mean of its square
  m_hat <- m / (1 - b1^t) # undo the bias from starting at zero
  u_hat <- u / (1 - b2^t)
  # Dividing by the root of the second moment gives each coordinate
  # its own step size.
  list(
    theta = theta - h * m_hat / (sqrt(u_hat) + eps),
    state = list(m = m, u = u)
  )
}

adam_start <- list(m = c(0, 0), u = c(0, 0))
cat("Adam:                ", updates_needed(adam, adam_start), "\n")
```

Adam: 82

Adam also needs fewer updates than plain gradient descent^a on this cost. **Adam** helps by dividing each coordinate's step by the typical size of its own gradient, so the shallow coordinate is no longer slowed by its small gradient and every coordinate moves by roughly the same amount per update: the size of each step is bounded by about α , whatever the scale of the gradient [24, Sec. 2.1]. In effect Adam is a cheap diagonal^b preconditioner [24, Sec. 5], which does for the gradient something like what rescaling the features does for the data. Neither changes the cost or the minimum, only the path taken toward it, and each adds hyperparameters (β , or β_1 , β_2 and ϵ) to tune.

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

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

Example 12.4 (Plain descent, momentum and Adam on the same objective). Let:

- n be the number of updates;
- α_{plain} , α_{mom} and α_{adam} be the **learning rates** of plain descent, **momentum** and **Adam**;
- β be the **momentum coefficient**, and β_1 , β_2 and ϵ the Adam settings.

In the **Auto** data [17], minimize the sample mean squared error (sMSE) of **mpg** on **horsepower** (in hundreds) and **weight** (in thousands), starting from $\vec{\beta} = \vec{0}$. Those two features are strongly correlated and sit far from zero, which makes the objective a long, narrow valley. Run $n = 200$ updates of plain gradient descent^a ($\alpha_{\text{plain}} = 0.08$, just under the largest step that does not diverge), of momentum ($\alpha_{\text{mom}} = 0.01$, $\beta = 0.9$), and of Adam ($\alpha_{\text{adam}} = 2$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$), and report the

sMSE reached by each, against the least-squares minimum:

12.11 Python

```
import numpy as np
import pandas as pd

auto_ma = pd.read_csv("data/auto.csv").dropna(subset=["horsepower"])
# Rescale to hundreds of horsepower and thousands of pounds, without centering.
Z_ma = auto_ma[["horsepower", "weight"]].to_numpy(float) / [100, 1000]
X_ma = np.column_stack([np.ones(len(Z_ma)), Z_ma])
y_ma = auto_ma["mpg"].to_numpy(float)
n = int(r.n_ma)

def cost_ma(b):
    return np.mean((X_ma @ b - y_ma) ** 2)

def grad_ma(b):
    return 2 * X_ma.T @ (X_ma @ b - y_ma) / len(y_ma)

# Plain gradient descent.
b = np.zeros(3)
for _ in range(n):
    b = b - r.h_plain_ma * grad_ma(b)
plain_ma = cost_ma(b)

# Momentum: step along a running velocity instead of the raw gradient.
b, v = np.zeros(3), np.zeros(3)
for _ in range(n):
    v = r.beta_ma * v + grad_ma(b)
    b = b - r.h_mom_ma * v
momentum_ma = cost_ma(b)

# Adam: moving averages of the gradient and its square, corrected for their start-up bias.
b, m, u = np.zeros(3), np.zeros(3), np.zeros(3)
for t in range(1, n + 1):
    g = grad_ma(b)
    m = r.beta1_ma * m + (1 - r.beta1_ma) * g
    u = r.beta2_ma * u + (1 - r.beta2_ma) * g ** 2
    m_hat = m / (1 - r.beta1_ma ** t)
    u_hat = u / (1 - r.beta2_ma ** t)
    b = b - r.h_adam_ma * m_hat / (np.sqrt(u_hat) + r.eps_ma)
adam_ma = cost_ma(b)

best_ma = cost_ma(np.linalg.lstsq(X_ma, y_ma, rcond=None)[0])
# Plain descent diverges once h exceeds 1 / (the largest eigenvalue of X'X / n).
h_max_ma = 1 / np.linalg.eigvalsh(X_ma.T @ X_ma / len(y_ma)).max()
print(round(plain_ma, 3), round(momentum_ma, 3), round(adam_ma, 3), round(best_ma, 3), round(h_max_ma, 3))
19.523 18.287 17.873 17.841 0.0853
```

12.12 R

```

auto_ma <- na.omit(read.csv("data/auto.csv")[, c("mpg", "horsepower", "weight")])
# Rescale to hundreds of horsepower and thousands of pounds, without centering.
X_ma <- cbind(1, as.matrix(auto_ma[, c("horsepower", "weight")]) %*% diag(1 / c(100, 1000)))
y_ma <- auto_ma$mpg

cost_ma <- function(b) mean((X_ma %*% b - y_ma)^2)
grad_ma <- function(b) drop(2 * t(X_ma) %*% (X_ma %*% b - y_ma) / length(y_ma))

# Plain gradient descent.
b <- rep(0, 3)
for (i in seq_len(n_ma)) b <- b - h_plain_ma * grad_ma(b)
plain_ma_r <- cost_ma(b)

# Momentum: step along a running velocity instead of the raw gradient.
b <- rep(0, 3)
v <- rep(0, 3)
for (i in seq_len(n_ma)) {
  v <- beta_ma * v + grad_ma(b)
  b <- b - h_mom_ma * v
}
momentum_ma_r <- cost_ma(b)

# Adam: moving averages of the gradient and its square, corrected for their start-up bias.
b <- rep(0, 3)
m <- rep(0, 3)
u <- rep(0, 3)
for (t in seq_len(n_ma)) {
  g <- grad_ma(b)
  m <- beta1_ma * m + (1 - beta1_ma) * g
  u <- beta2_ma * u + (1 - beta2_ma) * g^2
  m_hat <- m / (1 - beta1_ma^t)
  u_hat <- u / (1 - beta2_ma^t)
  b <- b - h_adam_ma * m_hat / (sqrt(u_hat) + eps_ma)
}
adam_ma_r <- cost_ma(b)

best_ma_r <- cost_ma(coef(lm(y_ma ~ X_ma[, -1])))
# Plain descent diverges once h exceeds 1 / (the largest eigenvalue of X'X / n).
h_max_ma_r <- 1 / max(eigen(crossprod(X_ma) / nrow(X_ma))$values)
round(c(plain = plain_ma_r, momentum = momentum_ma_r, adam = adam_ma_r, least_squares = best_ma_r), 3)

```

plain	momentum	adam	least_squares
19.523	18.287	17.873	17.841

After $n = 200$ updates, plain descent has reached an sMSE of 19.52, **momentum** 18.29 and **Adam** 17.87, against the minimum of 17.84. Plain descent cannot take a larger step than 0.085, because the steep direction across the valley would then make it diverge, and at that step size it crawls along the floor. Momentum gets closer by accumulating the steady component of the gradient along the valley, and Adam by scaling each coordinate's step by the size of its recent gradients. The ordering depends on the step sizes chosen, which were tuned for this objective.

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

Figure 12 shows plain descent and **momentum** on the cost $C(\vec{\beta}) = \frac{1}{2}(\beta_1^2 + 10\beta_2^2)$. That cost's condition number is 10.

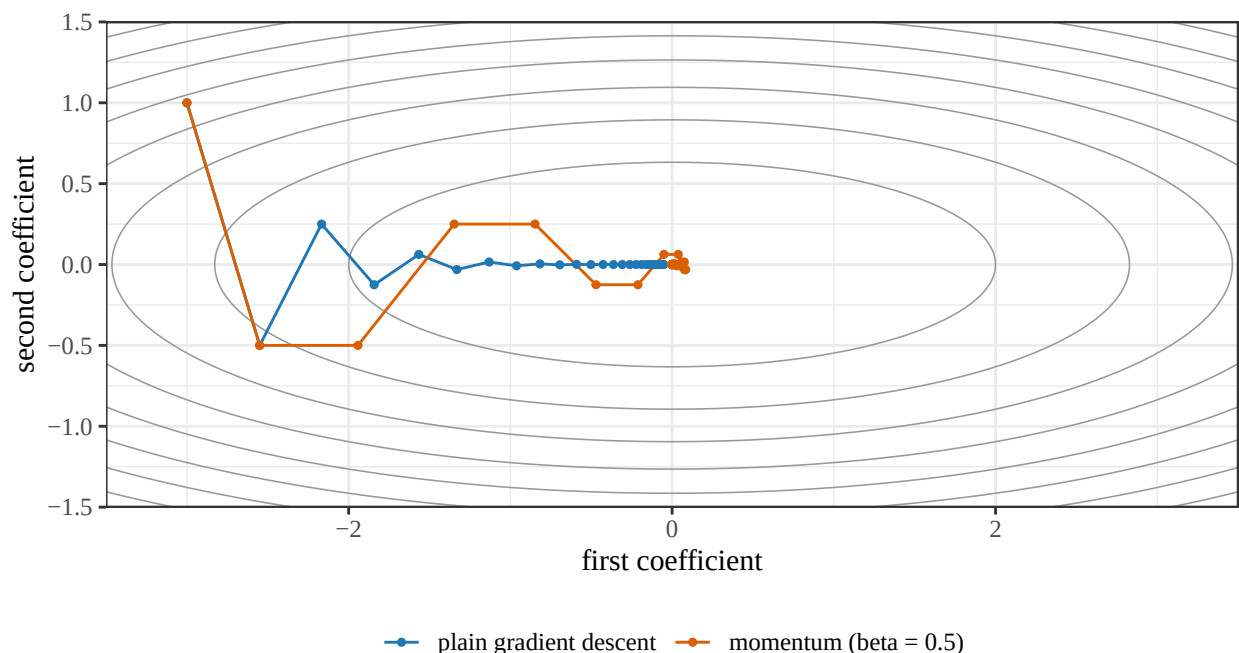


Figure 12: Plain gradient descent¹⁹ and heavy-ball momentum, with momentum coefficient $\beta = 0.5$, from $(-3, 1)$ on the quadratic cost $C(\vec{\beta}) = \frac{1}{2}(\beta_1^2 + 10\beta_2^2)$, both using step size 0.15 for 25 steps. Plain descent zig-zags across the steep vertical direction for its first few steps, as each step halves its vertical distance from the minimum and flips its side, and then creeps along the valley. Momentum still oscillates across the valley, but its accumulated velocity carries it faster along the valley floor, so it ends closer to the minimum after the same 25 steps. Grey curves are the cost's level sets.

13 In code

The same descent, three ways, on the three points of Exercise 3.1. Each block runs 200 full-batch steps at the optimal constant step size $\alpha^* = 1/8$ from Solution 7.13, checks the result against the closed-form solution, and then runs stochastic descent on the same data with the shrinking step size $\alpha_t = 1/(10 + t)$ that Solution 10.13 argues for.

The full-batch loop reaches the exact answer to six decimal places. The stochastic loop, after 20,000 single-point steps, is still only correct to about two — on a problem with three data points, where a full step costs three times what a stochastic one does. Stochastic descent is not a better optimizer. It is the one that remains affordable when n is large enough that the comparison stops being fair.

13.0.1 Python

```
import numpy as np

X = np.array([[1.0, 0.0], [1.0, 1.0], [1.0, 2.0]])
y = np.array([1.0, 3.0, 3.0])

def cost(beta):
    r = X @ beta - y
    return r @ r
```

```
def grad(beta):
    return 2 * X.T @ (X @ beta - y)

beta = np.zeros(2)
h = 1 / 8
for _ in range(200):
    beta = beta - h * grad(beta)

print(beta.round(6))

[1.333333 1.      ]
print(round(cost(beta), 6))

0.666667
print(np.allclose(beta, np.linalg.lstsq(X, y, rcond=None)[0]))
```

True

```
rng = np.random.default_rng(0)
beta_sgd = np.zeros(2)
for t in range(1, 20001):
    i = rng.integers(3)
    r = X[i] @ beta_sgd - y[i]
    beta_sgd = beta_sgd - (1 / (10 + t)) * 2 * r * X[i]

print(beta_sgd.round(2))
```

[1.33 1.]

13.0.2 R

```
X <- cbind(1, c(0, 1, 2)) # nolint: object_name_linter. X matches the prose.
y <- c(1, 3, 3)

cost <- function(beta) sum((X %*% beta - y)^2)
grad <- function(beta) as.vector(2 * t(X) %*% (X %*% beta - y))

beta <- c(0, 0)
h <- 1 / 8
for (i in 1:200) beta <- beta - h * grad(beta)

print(round(beta, 6))
```

[1] 1.333333 1.000000

```
print(round(cost(beta), 6))
```

[1] 0.666667

```
print(all.equal(beta, unname(coef(lm(y ~ X[, 2])))))
```

[1] TRUE

```
set.seed(0)
beta_sgd <- c(0, 0)
for (t in 1:20000) {
    i <- sample.int(3, 1)
    r <- sum(X[i, ] * beta_sgd) - y[i]
```

```

    beta_sgd <- beta_sgd - (1 / (10 + t)) * 2 * r * X[i, ]
}

print(round(beta_sgd, 2))

```

```
[1] 1.32 1.00
```

13.0.3 Julia

```

using LinearAlgebra, Random

X = [1.0 0.0; 1.0 1.0; 1.0 2.0];
y = [1.0, 3.0, 3.0];

cost(beta) = sum(abs2, X * beta - y);
grad(beta) = 2 * X' * (X * beta - y);

beta = zeros(2);
h = 1 / 8;
for _ in 1:200
    global beta -= h * grad(beta)
end

println(round.(beta, digits = 6))

```

```
[1.333333, 1.0]
```

```
println(round(cost(beta), digits = 6))
```

```
0.666667
```

```
println(isapprox(beta, X \ y))
```

```
true
```

```

Random.seed!(0);
beta_sgd = zeros(2);
for t in 1:20000
    i = rand(1:3)
    r = dot(X[i, :], beta_sgd) - y[i]
    global beta_sgd -= (1 / (10 + t)) * 2 * r * X[i, :]
end

println(round.(beta_sgd, digits = 2))

```

```
[1.32, 1.01]
```

The three random streams differ, so the stochastic answers differ in the last digit shown. Julia's default stream is not guaranteed stable across releases, so its printed value may move with the Julia version without anything in the code changing. What matches is the procedure, and within each block the full-batch fit agrees with the closed form exactly.

14 On real data

The three-point problem of Exercise 3.1 is small enough to check by hand. The next exercise runs the same descent on a real data set.

Exercise 14.1 (Gradient descent for fuel economy). The `Auto` data of James et al. [17] give the fuel efficiency (`mpg`) and engine horsepower of 392 cars. Fit the straight line $\text{mpg} \approx \beta_0 + \beta_1 z$, where z is horsepower standardized to mean 0 and standard deviation 1. Minimize the sample mean squared error (sMSE) with gradient descent from $\vec{\beta} = \vec{0}$, using the constant step sizes $\alpha = 0.05, 0.2$ and 0.9 for 40 steps each.

Which step size reaches the minimum cost fastest? How close is the line after 5 steps with the middle step size to the least-squares line?

Solution

Solution 14.1.

14.0.1 Python

```
import numpy as np
import pandas as pd

auto_py = pd.read_csv("data/auto.csv")[["mpg", "horsepower"]].dropna()
print(auto_py.head(3))

   mpg  horsepower
0  18.0         130
1  15.0         165
2  18.0         150

y_py = auto_py["mpg"].to_numpy()
hp = auto_py["horsepower"]
z_py = ((hp - hp.mean()) / hp.std()).to_numpy()
X_py = np.column_stack([np.ones(len(z_py)), z_py])

def descend_py(h, steps):
    beta = np.zeros(2)
    path = [beta]
    for _ in range(steps):
        beta = beta - h * 2 * X_py.T @ (X_py @ beta - y_py) / len(y_py)
        path.append(beta)
    return np.array(path)

paths_py = [descend_py(h, 40) for h in (0.05, 0.2, 0.9)]
beta_ols_py = np.linalg.lstsq(X_py, y_py, rcond=None)[0]
print("after 5 steps, h = 0.2:", paths_py[1][5].round(2))

after 5 steps, h = 0.2: [21.62 -5.6 ]

print("least squares:      ", beta_ols_py.round(2))

least squares:          [23.45 -6.08]
```

14.0.2 R

```
auto <- read.csv("data/auto.csv") |>
  dplyr::select(mpg, horsepower) |>
  tidyr::drop_na()
head(auto, 3)

# A tibble: 3 x 2
   mpg horsepower
<dbl>     <int>
1    18         130
```

```

2    15    165
3    18    150

z <- as.vector(scale(auto$horsepower))
X <- cbind(1, z) # nolint: object_name_linter. X matches the prose.
y <- auto$mpg

cost <- function(beta) mean((X %*% beta - y)^2)
grad <- function(beta) as.vector(2 * t(X) %*% (X %*% beta - y) / length(y))

descend <- function(h, steps) {
  beta <- c(0, 0)
  path <- matrix(NA_real_, nrow = steps + 1, ncol = 2)
  path[1, ] <- beta
  for (i in seq_len(steps)) {
    beta <- beta - h * grad(beta)
    path[i + 1, ] <- beta
  }
  path
}

step_sizes <- c(0.05, 0.2, 0.9)
paths <- lapply(step_sizes, descend, steps = 40)
beta_ols <- unname(coef(lm(y ~ z)))

```

After 40 steps, the smallest step size is still 0.13 above the minimum cost, while the other two are within 0.00001 of it. After 5 steps with the middle step size, the intercept is 21.62 and the slope is -5.6, against 23.45 and -6.08 for least squares. The dashed line in the right panel of Figure 13 is already close to the solid one, and slightly flatter.

i Remark

Remark 14.1. The cost is a quadratic whose Hessian^a is $\frac{2}{n}D^\top D$. It does not depend on mpg. Constant-step descent converges only when $\alpha < 2/L$, where L is the largest eigenvalue of that Hessian. Here $L = 2$, so the bound is $\alpha < 1$. All three step sizes in Figure 13 are below it. The largest, $\alpha = 0.9$, is close to the bound, so each step overshoots the minimum and lands on the other side, and the cost still falls, but more slowly than for $\alpha = 0.2$.

^a<https://morrison-lab.github.io/mds/vector-calculus.html#def-hessian>

```

library(ggplot2)
library(patchwork)
cost_df <- tibble::tibble(
  step = rep(0:40, times = length(step_sizes)),
  cost = as.vector(sapply(paths, function(p) apply(p, 1, cost))),
  h = factor(rep(step_sizes, each = 41), labels = paste("h =", step_sizes))
)
p_cost <- ggplot(cost_df, aes(step, cost, colour = h)) +
  geom_line() +
  scale_y_log10() +
  labs(x = "step", y = "mean squared error", colour = NULL) +
  theme(legend.position = "bottom")

line_df <- tibble::tibble(
  intercept = c(beta_ols[1], paths[[2]][6, 1]),
  slope = c(beta_ols[2], paths[[2]][6, 2]),
  line = c("least squares", "after 5 steps")
)

```

```

)
# The fit uses standardized horsepower; the ticks show horsepower.
hp_ticks <- pretty(auto$horsepower)
hp_mean <- mean(auto$horsepower)
hp_sd <- sd(auto$horsepower)
p_fit <- ggplot(tibble::tibble(z = z, y = y), aes(z, y)) +
  geom_point(colour = "grey60") +
  geom_abline(aes(intercept = intercept, slope = slope,
                  colour = line, linetype = line),
              data = line_df, linewidth = 0.9) +
  scale_colour_manual(values = c("least squares" = "black",
                                "after 5 steps" = "#00BA38")) +
  scale_linetype_manual(values = c("least squares" = "solid",
                                   "after 5 steps" = "dashed")) +
  scale_x_continuous(
    breaks = (hp_ticks - hp_mean) / hp_sd, labels = hp_ticks
  ) +
  labs(x = "horsepower", y = "mpg",
       colour = NULL, linetype = NULL) +
  theme(legend.position = "bottom")

p_cost + p_fit

```

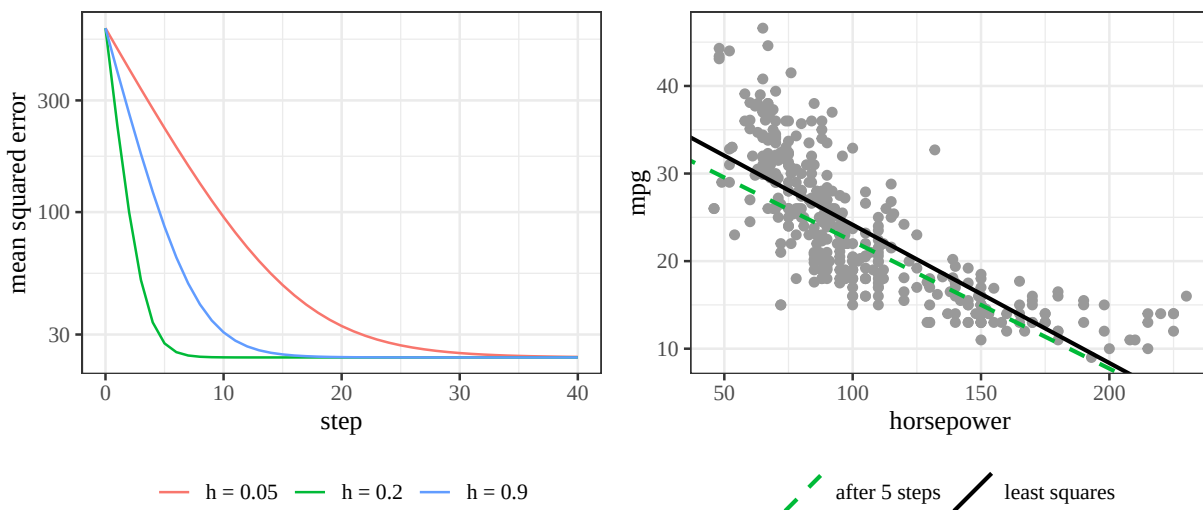


Figure 13: Gradient descent for the line relating mpg to horsepower, fitted on standardized horsepower. Left: the squared-error cost against the step number for three constant step sizes. Right: the data, the least-squares line, and the line after 5 steps with the middle step size.

Figure 14 plots the cost against the iteration for the three [step sizes](#) of Exercise 14.1.

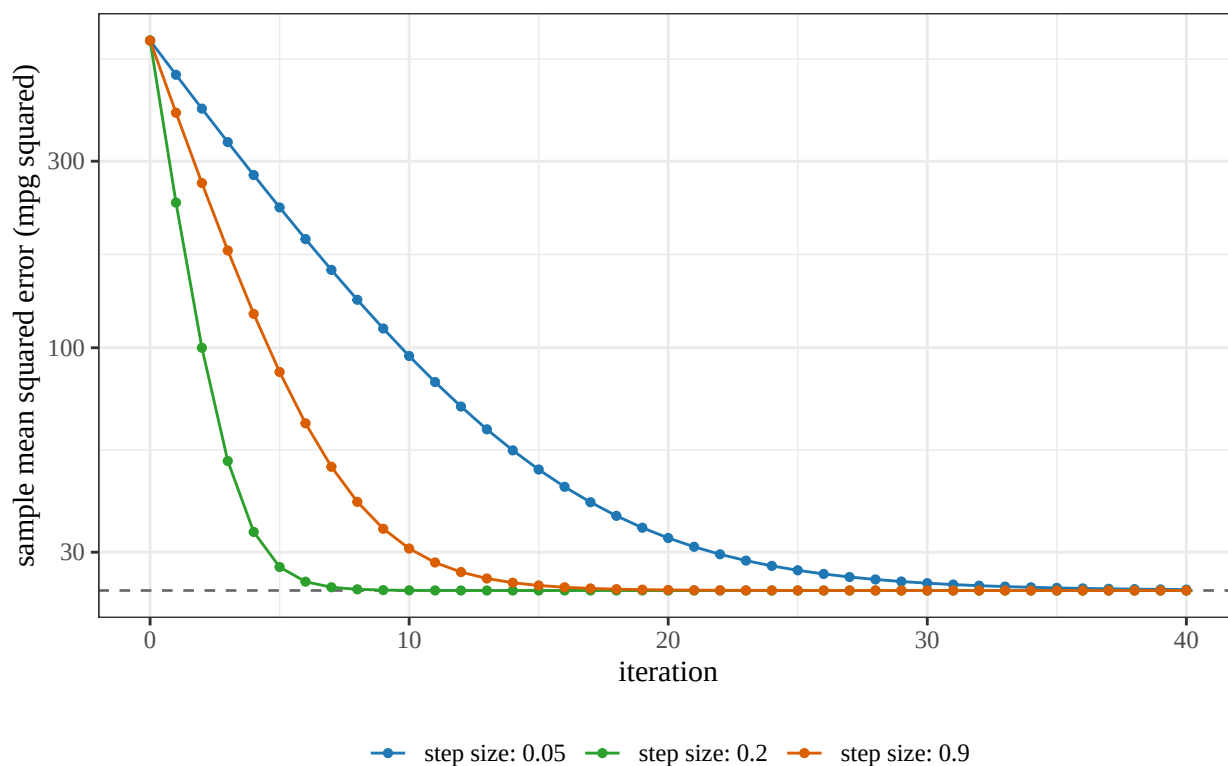


Figure 14: Sample mean squared error on a logarithmic scale across forty iterations of gradient descent²⁰ fitting miles per gallon on standardized horsepower in the Auto data from $(0, 0)$. A [step size](#) of 0.2 reaches the dashed least-squares minimum quickly. A smaller step size of 0.05 makes slow progress, while a large step size of 0.9 nears the stability limit and loses speed to oscillation.

15 Where this goes next

Gradient descent, or a variant of it, fits many of the models in the rest of these notes. Three topics continue this chapter directly:

- **Logistic Regression** is where these notes first take these steps on a model with no closed form. Its gradient, $\sum_i (\text{expit}(\vec{\beta} \cdot \tilde{x}_i) - y_i) \tilde{x}_i$, has a shape worth comparing with the least-squares gradient: a badly wrong prediction contributes at most $\|\tilde{x}_i\|$ to it, where squared error contributes without bound. See Logistic Regression.
- **Neural networks** are fitted by exactly the algorithm on this page, with two additions. Backpropagation is how ∇C is computed when the model is a composition of many layers, and the cost is no longer convex, so the guarantees of Exercise 7.1 do not apply — there are many local minima, and which one we reach depends on $\vec{\beta}_0$. See Neural Networks.

Watch

Sanderson shows gradient descent training a neural network, from 6:55^a [25, Ch. 2].

^a<https://www.youtube.com/watch?v=IHZwWFHWa-w&t=415s>

- **The modern optimizers** — momentum, AdaGrad, RMSProp, Adam, built in [Momentum and adaptive step sizes](#) — are all modifications of the update rule in [The algorithm](#), and all of them are attempts to reduce the effect of the condition-number problem of Exercise 7.10 without computing any eigenvalues. Momentum adds a fraction of the previous step to the current one, which damps the zig-zag; the adaptive

methods give each coordinate its own step size, which is an inexpensive substitute for rescaling the features.

16 Further reading

- Boyd and Vandenberghe [26], Section 9.3, on the gradient method and its convergence
- Goodfellow et al. [19], Section 4.3
- Zhou [27] and Zhou [28] — Biostat M257 slides on Newton steps, quasi-Newton methods (BFGS and limited-memory BFGS), and Fisher scoring; the Optimization²¹ page of the math notes develops the optimization methods with proofs
- Robbins and Monro [13] — the stochastic approximation method that stochastic gradient descent descends from
- Starmer [29], on gradient descent — an illustrated walk-through with little algebra

i Watch

Andrew Ng’s lectures for DeepLearning.AI cover the methods used to train neural networks: mini-batches [30]; how the batch size matters [31]; momentum [32]; RMSProp [33]; Adam [34]; and learning-rate decay [35].

- [1] A. Ng *et al.*, “CS229: Machine learning, autumn 2018.” Stanford Online, lecture videos on YouTube, 2018. Available: <https://www.youtube.com/playlist?list=PLoROMvodv4rMiGQp3WXShtMGzqpFVfbU>
- [2] K. Weinberger, “CS4780: Machine learning for intelligent systems, spring 2017.” Cornell University, lecture videos on YouTube, 2017. Available: <https://www.youtube.com/playlist?list=PLl8OIHZGYOQ7bkVbuRthEsaLr7bONzbXS>
- [3] Wikipedia contributors, “Rosenbrock function — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Rosenbrock_function
- [4] J. Starmer, “Gradient descent, step-by-step.” StatQuest, video on YouTube, 2019. Available: <https://www.youtube.com/watch?v=sDv4f4s2SB8>
- [5] 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/
- [6] GeeksforGeeks, “Gradient descent algorithm in machine learning.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/gradient-descent-algorithm-and-its-variants/>
- [7] N. Jaques, “CSE 446/546: Machine learning, lecture slides.” University of Washington, 2026. Accessed: Oct. 12, 2026. [Online]. Available: <https://courses.cs.washington.edu/courses/cse446/26sp/schedule/lectures/>
- [8] Wikipedia contributors, “Attractor — Wikipedia, the free encyclopedia.” 2026. Available: <https://en.wikipedia.org/wiki/Attractor>
- [9] C. M. Bishop, *Pattern recognition and machine learning*. in Information science and statistics. New York: Springer, 2006.
- [10] Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio, “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization,” in *Advances in neural information processing systems*, 2014. Available: <https://arxiv.org/abs/1406.2572>
- [11] GeeksforGeeks, “Learning rate in neural network.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/impact-of-learning-rate-on-a-model/>
- [12] Wikipedia contributors, “Stochastic gradient descent — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Stochastic_gradient_descent

²¹<https://morrison-lab.github.io/mds/optimization.html>

- [13] H. Robbins and S. Monro, “A stochastic approximation method,” *The Annals of Mathematical Statistics*, vol. 22, no. 3, pp. 400–407, 1951, doi: 10.1214/aoms/1177729586²².
- [14] GeeksforGeeks, “Stochastic gradient descent (SGD).” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/ml-stochastic-gradient-descent-sgd/>
- [15] J. Starmer, “Stochastic gradient descent, clearly explained.” StatQuest, video on YouTube, 2019. Available: <https://www.youtube.com/watch?v=vMh0zPT0tLI>
- [16] GeeksforGeeks, “Mini-batch gradient descent with Python.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/ml-mini-batch-gradient-descent-with-python/>
- [17] 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/>
- [18] GeeksforGeeks, “Epoch in machine learning.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/epoch-in-machine-learning/>
- [19] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT Press, 2016. Available: <https://www.deeplearningbook.org/>
- [20] I. Loshchilov and F. Hutter, “SGDR: Stochastic gradient descent with warm restarts,” in *International conference on learning representations*, 2017. Available: <https://arxiv.org/abs/1608.03983>
- [21] 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>
- [22] GeeksforGeeks, “Learning rate decay.” GeeksforGeeks, tutorial article, 2025. Available: <https://www.geeksforgeeks.org/machine-learning/learning-rate-decay/>
- [23] Wikipedia contributors, “Gradient descent — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Gradient_descent
- [24] D. P. Kingma and J. Ba, “Adam: A method for stochastic optimization,” in *3rd international conference on learning representations*, 2015. doi: 10.48550/arXiv.1412.6980²³.
- [25] 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
- [26] S. Boyd and L. Vandenberghe, *Convex optimization*. Cambridge University Press, 2004. doi: 10.1017/CBO9780511804441²⁴.
- [27] H. Zhou, “Newton’s method and Fisher scoring.” Lecture notes for Biostat M257, Statistical Computing, University of California, Los Angeles, 2025. Available: <https://ucla-biostat-257.github.io/2025spring/slides/23-newton/newton.html>
- [28] H. Zhou, “Quasi-Newton methods.” Lecture notes for Biostat M257, Statistical Computing, University of California, Los Angeles, 2025. Available: <https://ucla-biostat-257.github.io/2025spring/slides/24-quasinewton/quasinewton.html>
- [29] J. Starmer, *The StatQuest illustrated guide to machine learning*. StatQuest Publications, 2022. Available: <https://statquest.org/statquest-store/>
- [30] A. Ng, “Mini batch gradient descent.” DeepLearning.AI, Deep Learning Specialization, Course 2; video on YouTube, 2017. Available: <https://www.youtube.com/watch?v=4qJaSmvix8>
- [31] A. Ng, “Understanding mini-batch gradient descent.” DeepLearning.AI, Deep Learning Specialization, Course 2; video on YouTube, 2017. Available: https://www.youtube.com/watch?v=-_4Zi8fCZO4
- [32] A. Ng, “Gradient descent with momentum.” DeepLearning.AI, Deep Learning Specialization, Course 2; video on YouTube, 2017. Available: https://www.youtube.com/watch?v=k8fTYJPd3_I
- [33] A. Ng, “RMSProp.” DeepLearning.AI, Deep Learning Specialization, Course 2; video on YouTube, 2017. Available: https://www.youtube.com/watch?v=_e-LFe_igno
- [34] A. Ng, “Adam optimization algorithm.” DeepLearning.AI, Deep Learning Specialization, Course 2; video on YouTube, 2017. Available: https://www.youtube.com/watch?v=JXQT_vxqwIs
- [35] A. Ng, “Learning rate decay.” DeepLearning.AI, Deep Learning Specialization, Course 2; video on YouTube, 2017. Available: <https://www.youtube.com/watch?v=QzulmoOg2JE>

²²<https://doi.org/10.1214/aoms/1177729586>

²³<https://doi.org/10.48550/arXiv.1412.6980>

²⁴<https://doi.org/10.1017/CBO9780511804441>