

CS331: Algorithms and Complexity

Part VI: Continuous Algorithms

Kevin Tian

1 Introduction

In this part of the notes, we focus on algorithms for continuous optimization problems. Many algorithms we have studied so far in this course can naturally be phrased as optimization problems over a variable $\mathbf{x}$ living in some constraint set $\mathcal{X}$, i.e., for some $f : \mathcal{X} \rightarrow \mathbb{R}$, we want to solve

$$\min_{\mathbf{x} \in \mathcal{X}} f(\mathbf{x}) \text{ or } \max_{\mathbf{x} \in \mathcal{X}} f(\mathbf{x}). \quad (1)$$

We call $\mathbf{x}$ the *decision variable*. Essentially every problem in Parts III, IV, and V of the notes had the form (1) for an appropriate choice of $\mathcal{X}$ and f . For example, in scheduling, $\mathcal{X}$ was the set of all non-overlapping subsets of $[n]$, representing n intervals, and f was the subset size. Similarly, in s - t shortest path, $\mathcal{X}$ was the set of all s - t paths in a graph, and f was the total weight.

In all of these examples, $\mathcal{X}$ is discrete, i.e., we are asking for the best object out of a finite set of objects. However, we live in a continuous world, and many optimization problems in applications are naturally continuous, i.e., where $\mathcal{X}$ contains infinitely many points. This unit serves as a gentle introduction to techniques for approaching such problems. In the rest of these notes, the decision variable $\mathbf{x} \in \mathbb{R}^d$ will always be a vector, and $\mathcal{X} \subseteq \mathbb{R}^d$ will always be a continuous set.

To illustrate, consider the s - t maxflow problem on $G = (V, E, \mathbf{c})$, where the dimension $d = |E|$ is the edge count; we identify $E \equiv [d]$ in some order. The vector $\mathbf{x} \in \mathbb{R}^E$ represents a feasible s - t flow, and our goal is to solve the max variant of (1), where $f(\mathbf{x}) := \sum_{e=(s,v) \in E} \mathbf{x}_e$ and

$$\mathcal{X} := \left\{ \mathbf{x} \in \mathbb{R}_{\geq 0}^E \mid \mathbf{x} \leq \mathbf{c} \text{ entrywise, } \sum_{e=(v,u) \in E} \mathbf{x}_e - \sum_{e=(u,v) \in E} \mathbf{x}_e = 0 \text{ for all } v \in V \setminus \{s, t\} \right\}. \quad (2)$$

The problem of maximizing f over the continuous set (2), written in this way, appears quite daunting. However, in Section 4.3, Part V, we were nonetheless able to capitalize upon the structure of graphs to design algorithms for solving s - t maxflow. The purpose of these notes is to develop more general principles for solving structured continuous optimization problems.

The oracle model. Before designing algorithms, we should address an important issue: what sort of operations can we perform when solving (1)? Certainly being able to evaluate f at a given point $\mathbf{x}$ is a minimal requirement. In some natural applications, it is reasonable to assume that this is essentially the only form of access to f we have, e.g., when f is an extremely complicated function such as a deep neural network. If the function f is not too irregular, this also lets us approximate derivatives of f through finite differences: e.g., in one dimension,

$$f'(x) \approx \frac{f(x + \delta) - f(x)}{\delta} \quad (3)$$

for small δ , which can be computed in two function evaluations to f . In the more structured examples in these notes, where we have a more explicit description of the function f or constraint set $\mathcal{X}$, we will be afforded additional types of access to the problem.

Let us now give a simple example that demonstrates some pitfalls which can occur when performing continuous optimization in settings that are too unstructured.

Lemma 1. *Let $f : [0, 1] \rightarrow [0, 1]$. There is no algorithm that queries f at a finite number of points and returns a point $\hat{x} \in [0, 1]$ such that $f(\hat{x}) - \min_{x \in [0, 1]} f(x) \leq 0.99$.*

Proof. Consider a family of functions $f_{\bar{x}} : [0, 1] \rightarrow [0, 1]$, indexed by $\bar{x} \in [0, 1]$. We define $f_{\bar{x}}$ as:

$$f_{\bar{x}}(x) = \begin{cases} 0 & x = \bar{x} \\ 1 & x \neq \bar{x} \end{cases}.$$

Suppose f is chosen to be one of the $f_{\bar{x}}$ for an unknown $\bar{x}$. If our algorithm has found an $\hat{x}$ that satisfies $f(\hat{x}) - \min_{x \in [0, 1]} f(x) \leq 0.99$, then we must have $\hat{x} = \bar{x}$, i.e., we have learned the value of $\bar{x}$. However, this is a contradiction; given any finite set of function queries, there are an infinite number of $\bar{x}$ that make all queries evaluate to 1, preventing us from learning $\bar{x}$. $\square$

At face value Lemma 1 seems quite discouraging. Even if the function f is one-dimensional, with domain and range bounded by 1, and even if our goal is only to produce a point whose value is within 0.99 of the optimum, (1) is still impossible. Nonetheless, we will see that in many interesting cases, that a priori may seem even more challenging than the problem in Lemma 1, we can give efficient algorithms. The key challenge encountered in Lemma 1 was the lack of information; querying suboptimal points gives no clues as to where the minimizer lies.

When can we do better? A simple example is (1-d) *unimodal functions*, i.e., functions $f : [a, b] \rightarrow \mathbb{R}$ which are decreasing over $[a, c]$ and increasing over $[c, b]$, for some $a \leq c \leq b$ (so the minimizer is c). Here, a strategy called ternary search lets us quickly shrink the range in which the minimizer can lie by a constant factor each iteration. More concretely, assuming we know a candidate range $[\ell, r] \subset [a, b]$ containing c , querying $f(\frac{2\ell}{3} + \frac{r}{3})$ and $f(\frac{\ell}{3} + \frac{2r}{3})$ lets us eliminate a third of the candidate range, depending on which query is larger, upon which we can recurse.

The key difference between these two examples is that in the latter case, we were able to use function query information to locally improve and hone in on the minimizer. Indeed, *convexity*, a variant of unimodality which extends to high dimensions, will be a recurring theme in these notes, as an example of structure that gives way to tractability in continuous optimization.

We remark that the author of these notes developed a new course on continuous algorithms. The interested reader can find a much more in-depth discussion of any topic in these notes at [Tia24a].

2 Statistics and machine learning

The language of continuous optimization is the language of modern algorithms for statistical estimation and machine learning (ML). More broadly, there has been a longstanding interplay between statistics, ML, and optimization as fields. In this section, we provide a high-level overview of this connection, and how continuous optimization drives the training of modern ML systems.

2.1 Statistical inference

Statistics concerns the following setting: there is an underlying distribution D over some space $\mathcal{S}$, and we receive samples $\{s^{(1)}, \dots, s^{(n)}\} \in \mathcal{S}^n$ from D . Our goal is to infer something about the underlying distribution D from the set of $\{s^{(1)}, \dots, s^{(n)}\} \in \mathcal{S}^n$ (the “dataset”). This problem setting is completely ubiquitous to modeling how we understand how our world works, and captures essentially any setting where something we care about (“the underlying truth”) is passed through a noisy channel (“the distribution”), and cannot be easily recovered.

Perhaps the simplest example is in the *binary outcome* setting, where the only observable outcomes $\mathcal{S}$ are yes-no answers, or $s \in \{0, 1\}$. Here, the underlying distribution D is a *Bernoulli distribution* $\text{Bern}(p^*)$, i.e., for some unknown $p^* \in [0, 1]$, a sample from D is set to 1 with probability p^* , and is set to 0 otherwise. One common inference task is to observe draws from $D = \text{Bern}(p^*)$ and to estimate what p^* is. For example, given flips of a coin, is the coin fair ($p^* = \frac{1}{2}$) or loaded ($|p^* - \frac{1}{2}| \gg 0$)? This exact estimation task arises from models of many real-world events, such as predicting the chance of rain on a given day, the chance that a particular team will win a sports game, or the chance that a randomized science experiment will yield the preferred outcome.

How do we evaluate whether we solved a particular statistical inference problem? A typical strategy is to parameterize the task as estimating a particular quantity $\mathbf{x}^* \in \mathbb{R}^d$. We evaluate the quality of our estimate, $\hat{\mathbf{x}} \in \mathbb{R}^d$, using a *loss function* $L : \mathbb{R}^d \rightarrow \mathbb{R}$. The goal is for higher-quality estimates

$\hat{\mathbf{x}}$, i.e., estimates that actually reflect D as desired, to yield smaller loss values. For example, in our running example where $D = \text{Bern}(p^*)$, a reasonable loss for evaluating $\hat{p} \in \mathbb{R}$ is

$$L(\hat{p}) := |\hat{p} - p^*|. \quad (4)$$

We make a few observations about the loss (4). First, it models our goal, in that the loss minimizer is $\hat{p} = p^*$, so successfully minimizing L actually does capture all relevant information about the underlying model D . Second, it is not a realistic loss because we cannot evaluate it unless we already knew p^* , which was the entire point of the estimation problem!

Case study: parametric statistics. The circular logic concerning (4) reflects a broader phenomenon in designing loss functions. On the one hand, the loss must somehow depend on the dataset $\{s^{(1)}, \dots, s^{(n)}\}$ we receive from D , because otherwise the produced estimate will be completely independent of D . On the other hand, to be useful to us in settings where D is not explicitly accessed except through observations, it can *only depend* on the dataset, because any other information about D is destroyed during the sampling process, and is not known to us. Thus, any realistic loss function L is inherently only a partial reflection of the truth.

Nevertheless, there is a common setting where coming up with a reasonable data-dependent loss function is straightforward: *parametric statistical estimation*. Here, the underlying distribution D really is completely captured by a *parameter*, taken as $\mathbf{x}^* \in \mathbb{R}^d$. Thus, to infer what D is, it suffices to recover $\mathbf{x}^*$. A simple example is our coin setting from earlier, where $D = \text{Bern}(p)$ is completely determined by $p \in \mathbb{R}$. Another fundamental example is mean estimation in *isotropic Gaussians*, where $D = \mathcal{N}(\mathbf{x}^*, \mathbf{I}_d)$ is a multivariate Gaussian (normal distribution). Here, a sample $\mathbf{s} \sim D$ has each coordinate $j \in [d]$ distributed as a variance-1 Gaussian centered at $\mathbf{x}_j^*$, i.e.,

$$\mathbf{s}_j \sim \mathcal{N}(\mathbf{x}_j^*, 1)$$

are observed independently at random. An important thing about Gaussians is that they have a closed-form *likelihood* density, i.e., we know that the probability density function (PDF) of $s \in \mathbb{R}$ under a Gaussian $\mathcal{N}(\mu, 1)$ is proportional to

$$\exp\left(-\frac{1}{2}(s - \mu)^2\right). \quad (5)$$

In other words, the further s is from μ (so the larger $(s - \mu)^2$ is), the less likely we will observe s from $\mathcal{N}(\mu, 1)$, consistently with our intuition. Similarly, the likelihood density for $\mathbf{s} \sim \mathcal{N}(\mathbf{x}^*, \mathbf{I}_d)$ is, by independence of the coordinates $\mathbf{s}_i$, proportional to

$$\exp\left(-\frac{1}{2}(\mathbf{s}_1 - \mathbf{x}_1^*)^2\right) \cdot \dots \cdot \exp\left(-\frac{1}{2}(\mathbf{s}_d - \mathbf{x}_d^*)^2\right) = \exp\left(-\frac{1}{2}\|\mathbf{s} - \mathbf{x}^*\|_2^2\right), \quad (6)$$

where we recall $\|\mathbf{v}\|_2^2 = \sum_{i \in [d]} \mathbf{v}_i^2$ is the squared length of a vector $\mathbf{v} \in \mathbb{R}^d$, by the Pythagorean theorem. Thus, $\mathbf{s} \sim \mathcal{N}(\mathbf{x}^*, \mathbf{I}_d)$ is more likely to be observed the shorter $\mathbf{s} - \mathbf{x}^*$ is. In particular, we expect “typical” samples to concentrate near the true mean $\mathbf{x}^*$ we wish to estimate.

In parametric settings where we have a closed-form expression for the likelihood of producing a dataset if the *true* parameter was $\hat{\mathbf{x}}$, it is reasonable to use a variant of this likelihood itself as a loss. To explain how this typically works, consider an observed dataset $\{\mathbf{s}^{(i)}\}_{i \in [n]} = \{\mathbf{s}^{(1)}, \dots, \mathbf{s}^{(n)}\}$ where each sample is independently drawn as $\mathbf{s}^{(i)} \sim \mathcal{N}(\mathbf{x}, \mathbf{I}_d)$. Based on the expression for the single-sample likelihood density (6), the *joint density* over all of the samples is

$$\prod_{i \in [n]} \exp\left(-\frac{1}{2}\|\mathbf{s}^{(i)} - \mathbf{x}\|_2^2\right) = \exp\left(-\frac{1}{2} \sum_{i \in [n]} \|\mathbf{s}^{(i)} - \mathbf{x}\|_2^2\right),$$

if the true mean was $\mathbf{x}$. The *maximum likelihood estimation* (MLE) approach simply takes this likelihood as a function of $\mathbf{x}$, and tries to maximize it.

This approach can be reinterpreted through the lens of loss minimization. Indeed, consider parameterizing the joint density of the observed samples as

$$\exp\left(-L(\mathbf{x} \mid \{\mathbf{s}^{(i)}\}_{i \in [n]})\right), \quad L(\mathbf{x} \mid \{\mathbf{s}^{(i)}\}_{i \in [n]}) := \frac{1}{2} \sum_{i \in [n]} \|\mathbf{s}^{(i)} - \mathbf{x}\|_2^2. \quad (7)$$

Here, $L(\mathbf{x} \mid \{\mathbf{s}^{(i)}\}_{i \in [n]})$ is the joint *negative log-likelihood (NLL)*, a function of $\mathbf{x}$ that evaluates how likely it was to observe the particular dataset we received. Then the MLE approach equivalently takes the NLL $L(\cdot \mid \{\mathbf{s}^{(i)}\}_{i \in [n]})$ to be our optimization objective for evaluating a candidate $\hat{\mathbf{x}}$. Indeed, the lower the NLL, the higher the actual likelihood is (since $-\log$ is a decreasing function), so the minimizer to the NLL is also just the maximum likelihood estimate. In general, the dataset generating process is noisy, and there is no guarantee that the MLE $\hat{\mathbf{x}} := \arg \min L(\cdot \mid \{\mathbf{s}^{(i)}\}_{i \in [n]})$ will actually be the true underlying $\mathbf{x}^*$. Much of statistical estimation theory considers how many samples n are required to achieve a particular level of estimation error.

The MLE philosophy for designing losses can be captured via a diagram:

$$\underbrace{\mathbf{x}^*}_{\text{unknown parameter}} \longrightarrow \underbrace{\{\mathbf{s}^{(i)}\}_{i \in [n]}}_{\text{dataset}} \longrightarrow \underbrace{\hat{\mathbf{x}}}_{\text{MLE}}. \quad (8)$$

The first arrow above captures the data generation process, whose likelihood is proportional to

$$\exp\left(-L(\mathbf{x} \mid \{\mathbf{s}^{(i)}\}_{i \in [n]})\right),$$

where $L(\cdot \mid \{\mathbf{s}^{(i)}\}_{i \in [n]})$ is the NLL. Then the second arrow in (8) corresponds to our estimation strategy, in this case returning the minimizer to the NLL:

$$\hat{\mathbf{x}} \leftarrow \arg \min_{\mathbf{x}} L(\mathbf{x} \mid \{\mathbf{s}^{(i)}\}_{i \in [n]}).$$

This estimator has the following interpretation: we pick the estimated parameter $\hat{\mathbf{x}}$ that is as likely as possible to have produced the particular dataset we observed, had it been the underlying parameter $\mathbf{x}^*$ in (8). Notice that the estimator $\hat{\mathbf{x}}$ can be computed without knowledge of the unknown parameter $\mathbf{x}^*$, because it only depends on the dataset. This solves our earlier circular logic, e.g., when designing the (uncomputable) loss function (4).

As a closing thought, simply defining the MLE is not an algorithm. Beyond stating that our strategy is to minimize L , to make this approach useful in real applications, we must also specify a way to actually minimize L . This question (and related questions) is the purpose of continuous algorithms. In particular, one lesson that is already apparent is the benefits of choosing a loss L that is “compatible” with optimizers, for example by avoiding pathologies such as Lemma 1.

Classical statistics as optimization. We give a few more examples of the MLE paradigm, which reframes statistical estimation as optimization (loss minimization).

Consider first a *Gaussian linear regression* model, where we believe that there is roughly a linear relationship between observed outcomes $y \in \mathbb{R}$ and known features $\mathbf{x} \in \mathbb{R}^d$. For example, the features could be various pieces of information about a person (e.g., their height, weight, age, etc.) and the observed outcome could be some measured statistic (e.g., the amount they spend on groceries in a week). In linear regression, the underlying belief is that there is a vector $\mathbf{w} \in \mathbb{R}^d$ that approximately governs the relationship between each $\mathbf{x}$ and its linked outcome y :

$$y \approx \langle \mathbf{w}, \mathbf{x} \rangle = \sum_{j \in [d]} \mathbf{w}_j \mathbf{x}_j.$$

Of course, due to observation noise or the effects of mismodeling (e.g., some feature that was important that we did not have access to), it is often unlikely that $y = \langle \mathbf{w}, \mathbf{x} \rangle$ exactly. In Gaussian linear regression, the error is modeled as Gaussian noise. In particular, each sample $\mathbf{s} = (\mathbf{x}, y)$ is generated as follows: an independent Gaussian $\xi \sim \mathcal{N}(0, 1)$ is drawn, and then we see

$$y = \langle \mathbf{w}, \mathbf{x} \rangle + \xi.$$

In this setting, the goal is to estimate $\mathbf{w}$ (the “best fit line”) from noisy samples $\{\mathbf{x}^{(i)}, y^{(i)}\}_{i \in [n]}$; notably, in the absence of ξ , this task is just solving a system of equations. By tracing back through the expression (5) for the likelihood of a single Gaussian sample, we can determine that the NLL for Gaussian linear regression is

$$L(\mathbf{w} \mid \{\mathbf{x}^{(i)}, y^{(i)}\}_{i \in [n]}) = \frac{1}{2} \sum_{i \in [n]} \left(\langle \mathbf{w}, \mathbf{x}^{(i)} \rangle - y^{(i)} \right)^2. \quad (9)$$

Minimizing loss functions with the structured form (9) is hence often called *linear regression*. We return to this example, and discuss algorithms for linear regression, later in these notes.

As a second example, we illustrate the difference between *regression* (predicting a real-numbered outcome) and *classification* (predicting a “category”, or one of k classes). The common binary classification setting takes $k = 2$ categories, and models the outcome $y \in \{0, 1\}$ as a nonlinear function of features $\mathbf{x}$. In such settings, a common model is to take a *link function* $\sigma : \mathbb{R} \rightarrow [0, 1]$ that maps a real value $\langle \mathbf{w}, \mathbf{x} \rangle$ to a probability in $[0, 1]$. Then, we observe a class

$$y = \begin{cases} 1 & \text{with probability } \sigma(\langle \mathbf{w}, \mathbf{x} \rangle) \\ 0 & \text{with probability } 1 - \sigma(\langle \mathbf{w}, \mathbf{x} \rangle) \end{cases}. \quad (10)$$

This can be modeled as the class y being drawn from a Bernoulli distribution $\text{Bern}(\sigma(\langle \mathbf{w}, \mathbf{x} \rangle))$, parameterized by the observed features $\mathbf{x}$, an unknown weight vector $\mathbf{w}$, and the link function σ . Again, our goal is to estimate $\mathbf{w}$, the hidden weights governing our observations. Logistic regression specifically considers this generative model with the link function

$$\sigma(t) := \frac{\exp(t)}{1 + \exp(t)}.$$

There are many strategies for estimating $\mathbf{w}$ in logistic regression, but MLE is a popular choice.

Finally, we mention that there are many extensions and generalizations to the setup discussed thus far, and that statistical estimation is a far-reaching paradigm. One example that builds on our logistic regression discussion is the problem of *clustering*, where we think of the dataset as arising from D , a *mixture model* of different components $\{D_1, \dots, D_k\}$, and we want to estimate which component each sample came from. The different components could correspond to different categories in a classification task, e.g., a mixture of different species in a collection of animal photos. Typically, we assume that there is a hidden parameter vector $\mathbf{p} \in [0, 1]^k$ with $\mathbf{p}_1 + \dots + \mathbf{p}_k = 1$ specifying the weights of the different classes. For each example $\mathbf{x}$, a class label $j \in [k]$ is first drawn according to $\mathbf{p}$, and then $\mathbf{x}$ is drawn from the component D_j . We can therefore think of $D = \sum_{j \in [k]} \mathbf{p}_j D_j$ as being a “weighted average” of its components.

Upon observing a dataset $\{\mathbf{x}^{(i)}\}_{i \in [n]}$ drawn from this mixture model, clustering asks to recover the *class labels*, i.e., to sort the samples into “similar” categories. This is a reasonable goal in settings where two samples from the same component D_j look more similar than samples from two different components. Notice that this problem appears inherently “discrete” optimization, in that the parameter we wish to recover is categorical. Nonetheless, because the underlying space on the samples $\{\mathbf{x}^{(i)}\}_{i \in [n]}$ is often continuous (e.g., $\mathbb{R}^d$), continuous optimization tools have been immensely useful in tackling clustering and other discrete estimation problems.

Other statistical inference tasks. Beyond producing point estimates in parametric statistics, there are many other interesting and practical problems studied by statistical inference. As a basic extension of our discussion, mean estimation is still meaningful in *nonparametric settings* where we do not make a parameterized assumption about D (e.g., that it is exactly a Gaussian), but still want to learn about a property of it. Even when the target density is not parametric, strategies derived from continuous loss minimization (such as returning the *empirical mean* $\frac{1}{n} \sum_{i \in [n]} \mathbf{s}^{(i)}$, which optimizes the Gaussian NLL (6)) are commonplace and often effective.

Statistical inference also encompasses a variety of other inherently algorithmic problems that are useful when dealing with random observations. This includes *hypothesis testing* (determining which proposed explanation best models observed reality), *uncertainty quantification* (finding sets that the true parameter lies in at a given confidence level, under appropriate data modeling assumptions), *causal inference* (deciding which features directly influence the observed outcome, and which are only spuriously correlated) and *experimental design* (choosing the best way to allocate data collection to draw accurate and credible conclusions). Despite seeming quite different than parametric statistical estimation, many of these other inference problems have also had a rich interplay with continuous optimization. For example, hypothesis testing can be viewed as choosing an optimal *decision rule* for picking between different candidate hypotheses, and determining the best decision boundaries is often an optimization problem in disguise.

2.2 Machine learning

The statistical viewpoint in Section 2.1, and its connections to continuous optimization, has been enormously influential in modern machine learning (ML). However, differences in the theory and practice of ML when compared to statistics have been shaped by the field’s somewhat distinct goals. In classical parametric statistics, the underlying parameter $\mathbf{x}^*$ is posited as being truly responsible for the generative process by which we receive our dataset $\{\mathbf{s}^{(1)}, \dots, \mathbf{s}^{(n)}\}$, and we wish to learn $\hat{\mathbf{x}} \approx \mathbf{x}^*$. On the other hand, ML’s goal is often more direct: we want to learn $\hat{\mathbf{x}}$ only insofar as it helps us make accurate future predictions. For example, a typical setup in *supervised learning* assumes that we have a *training dataset* $\{(\mathbf{x}^{(1)}, y^{(1)}), \dots, (\mathbf{x}^{(n)}, y^{(n)})\}$, consisting of n pairs of features $\mathbf{x}^{(i)} \in \mathbb{R}^d$ and their corresponding labels $y^{(i)}$. In classification tasks, the label $y^{(i)}$ simply indicates one of k classes. Our goal is to learn a predictor p that can take the features of a new example $\mathbf{x}$ that we do not have the label of, and output $p(\mathbf{x})$ that is an accurate guess for the true label. For example, we may train an ML predictor p using 1000 random images of dogs and cats that we have labels for, with the hope that it can output predictions $p(\mathbf{x}) \in \{0, 1\}$ corresponding to whether a new image $\mathbf{x}$ is a dog or a cat.

We pause to interpret this setup, and how it differs from statistical estimation. Firstly, the learned predictor p mirrors the role of the estimated parameter $\hat{\mathbf{x}}$ from Section 2.1. Just as statistics tries to evaluate $\hat{\mathbf{x}}$ based on how well it resembles the true (unknown) $\mathbf{x}^*$, ML similarly tries to evaluate p based on its performance on “unseen” data. This is typically formulated through the language of a *population loss*, where we define the loss to be

$$\mathbb{E}_{\mathbf{s} \sim D} [\ell(p \mid \mathbf{s})].$$

This expression indicates that the ideal evaluation of p is as follows: we draw a *fresh sample* $\mathbf{s} \sim D$ independently of p , and then evaluate the single-sample prediction loss $\ell(p \mid \mathbf{s})$. The overall population loss is the average such loss over the randomness of $\mathbf{s}$; in other words, it is the average performance of p on D . This criterion captures the difference between *memorization* and *learning*: in many cases, it is easy to minimize the empirical loss $\sum_{i \in [n]} \ell(p \mid \mathbf{s}^{(i)})$ on the samples, while failing to generalize (achieve low population loss). In ML, predictions are thought of as useful only when they successfully generalize, and much of the ecosystem surrounding the training of ML models (e.g., use of a *holdout test set* for evaluation) is built around this philosophy.

Secondly, although estimating a parameter sounds like a different goal than predicting future outcomes, they are actually the same goal if outcomes are determined by knowledge of some parameter! Our logistic regression example (10) from earlier is particularly reflective of this: if σ is known and the labels really are generated from (10), then given the features $\mathbf{x}$, the weight vector $\mathbf{w}$ contains all information we need to know to optimally guess the label y . Hence, a reasonable strategy in ML under data given by (10) is to simply estimate $\mathbf{w}$.

The key lesson of modern ML is that this philosophy continues to make sense (and be immensely useful), even if the parameter $\mathbf{w}$ we estimate is no longer truly responsible for the data we observed. As a simple example, even if some one-dimensional observations (e.g., test grades) were not literally generated by a Gaussian, if they satisfy similar properties, then Gaussian estimation techniques can still be useful in performing inference.¹ This is often referred to as the *agnostic learning* setting, where the true data-generating process does not belong to a parametric family captured by $\mathbf{w}$, but we nevertheless ask for the “best-fitting” member of this family to the data, and use it to make predictions. The agnostic learning setting contrasts with the *realizable learning* setting, where the data generative model is actually parametric. Just as in statistics, the best-fitting parametric model is computed by optimizing a continuous loss function $L(\mathbf{w} \mid \{\mathbf{s}^{(i)}\}_{i \in [n]})$.

Deep learning: the recipe. A particular instantiation of this recipe has proved enormously successful, and is largely responsible for the ML revolution of the last decade. Starting from the early 2010s, the ML research community discovered (as popularized by e.g., [KSH12]) that a particular class of powerful, extremely expressive parametric models could be combined with loss minimization to make accurate predictions across a startling range of domains.

More concretely, the recipe is instated as follows. Let $\mathbf{w} \in \mathbb{R}^d$ be the parameters of a particularly expressive family of models, e.g., deep neural networks. Let $p(\mathbf{x}; \mathbf{w})$ be the predictions made by

¹A highly relevant quote by George E. P. Box is “all models are wrong, but some are useful.”

a model with parameters $\mathbf{w}$. Often the range of p is probability distributions over k classes: for example, the range $[0, 1]$ is used to denote the probability of the positive class in binary classification. Assume also we have access to a labeled dataset of samples $\mathbf{s}^{(i)} = (\mathbf{x}^{(i)}, y^{(i)})$. Then we take $L(\mathbf{w} \mid \{\mathbf{s}^{(i)}\}_{i \in [n]})$ to be the total loss over training examples, of the prediction losses

$$\ell(p(\mathbf{x}^{(i)}), y^{(i)}),$$

for a simple function ℓ . Common choices in the binary classification setting ($y \in \{0, 1\}$) include

$$\ell(p, y) = \begin{cases} 0 & |y - p| \leq \frac{1}{2} \\ 1 & |y - p| > \frac{1}{2} \end{cases}, \quad \ell(p, y) = (p - y)^2, \quad \ell(p, y) = (1 - y) \log \left(\frac{1}{1 - p} \right) + y \log \left(\frac{1}{p} \right).$$

The first function above (the *zero-one loss*) is very intuitive: it simply assigns a loss of 1 to any prediction p that does not round to the actual outcome label. Unfortunately, due to its hard decision boundary, it is notoriously challenging to optimize robustly. The second and third functions, the *squared loss* and *cross entropy loss*, can be viewed as smoothed alternatives to the zero-one loss. The cross entropy loss, which is actually just the NLL for logistic regression (10), is in fact the predominant loss used in modern classification and language modeling tasks.

The remainder of the deep learning recipe amounts to choosing an expressive class parameterized by $\mathbf{w} \in \mathbb{R}^d$, and choosing an algorithm for optimizing the corresponding loss function

$$L(\mathbf{w}) := \sum_{i \in [n]} \ell(p(\mathbf{x}^{(i)}; \mathbf{w}), y^{(i)}).$$

Both problems have received enormous attention over the years. Regarding the latter question, an optimization algorithm called Adam [KB15] (and its weight decayed variant AdamW [LH19]) has emerged as the default choice, due to its fast and robust performance in practice and its ability to generalize well. Regarding the former question, i.e., architecture selection, many new deep learning architectures have been designed to address empirical observations when training on real-world data. This has led to the development of popular model classes such as convolutional neural networks [LBBH98], recurrent neural networks [Elm90, HS97], residual networks [HZRS16], and the *transformer* architecture [VSP⁺17] central to today’s frontier LLMs.

Other goals in ML. Supervised learning (in its classification or regression forms) is only one of several different important paradigms in modern ML. In supervised learning (as loss minimization over parameters), and other parametric estimation tasks, the goal is to estimate a property of a distribution D from examples. Supervised learning was often the focus of classical ML benchmarks, and the success of deep learning was fueled in large part by its success in supervised learning.

Another related goal is *generative modeling*, where the goal is to imitate the distribution D itself. For example, in diffusion models [HJA20, SSDK⁺21], learned trajectories for differential equations gradually transform white noise into new draws from the “sample distribution.” This allows the model to produce new examples “from D ,” as the generative model has learned, e.g., samples of new cat pictures from a diffusion model trained on a dataset of examples. Both large language models and image diffusion models can be naturally cast as generative models.

Finally, we mention *reinforcement learning* as yet another distinct paradigm in modern ML. In reinforcement learning, the learner is an agent interacting with its environment (such as a robot in a room), where the environment reacts to the agent’s choices. Thus, the learner’s actions can influence the samples it receives. Here, the goal is to learn a good *policy* (mapping from states to actions) that allows the agent to optimally manipulate its environment to succeed at its given task. The recipe for deep learning discussed thus far has also been influential in reinforcement learning, which seeks to answer questions such as: how do we parameterize good policies, and how do we model the reward? Perhaps unsurprisingly, deep neural networks are a powerful way to parameterize policies, and continuous optimization of a loss based on an apt reward function is an effective way to learn them [SWD⁺17]. Beyond its inherent value in problems best cast in the language of reinforcement learning (such as playing games, or having a robot move through a room), this paradigm has seen prominent use in *reinforcement learning from human feedback*, used to fine-tune LLMs using reward signals from human preferences [OWJ⁺22].

3 Gradient descent

In this section, we develop the most famous continuous algorithm: gradient descent (GD). Our focus in this section will be on motivating and analyzing GD in the one-dimensional setting,² but we briefly discuss high-dimensional variants in Section 3.4.

3.1 Structural assumptions

As illustrated in Lemma 1, unstructured continuous optimization is impossible, even when $\mathcal{X} \subseteq \mathbb{R}$ is one-dimensional. However, under *structural assumptions* on f that hold in important applications, it turns out that we can optimize f efficiently assuming nothing more than value query access, i.e., the ability to evaluate $f(x)$ at a given $x \in \mathcal{X}$.

Convexity. Perhaps the most natural structural assumption to impose on f to admit efficient optimization algorithms is *convexity*. We begin by defining this property.

Definition 1 (Convexity). *We say that a function $f : \mathbb{R} \rightarrow \mathbb{R}$ is convex if for any two $x, y \in \mathbb{R}$,*

$$f((1 - \lambda)x + \lambda y) \leq (1 - \lambda)f(x) + \lambda f(y), \text{ for all } \lambda \in [0, 1]. \quad (11)$$

We say f is strictly convex if (11) holds with strict inequality, i.e., $<$ rather than $\leq$.

Why is convexity helpful? One reason is that convex minimization is unambiguous; as you will explore in the homework, strictly convex functions are unimodal (see Section 1). Thus, all local minima are global minima, so we cannot be tricked into terminating an optimization algorithm.

Another major reason is that convexity implies helpful bounds, which can guide algorithms and measure their progress. For example, (11) has a simple interpretation as providing a function *upper bound*. Indeed, it states that f lies below the line between $(x, f(x))$ and $(y, f(y))$, over the interval $[x, y]$. We can further rearrange (11) to obtain the *lower bound*

$$f(y) \geq \frac{f((1 - \lambda)x + \lambda y) - (1 - \lambda)f(x)}{\lambda} = f(x) + \frac{f(x + \lambda(y - x)) - f(x)}{\lambda}.$$

If f is differentiable, then by taking a limit as $\lambda \rightarrow 0$,

$$f(y) \geq f(x) + \lim_{\lambda \rightarrow 0} \frac{f(x + \lambda(y - x)) - f(x)}{\lambda} = f(x) + f'(x)(y - x). \quad (12)$$

You may recall from calculus that the Euler approximation to a function, centered at x , is

$$f(y) \approx f(x) + f'(x)(y - x). \quad (13)$$

The consequence (12) has a simple interpretation: for convex f , the Euler approximation is always an underestimate. One implication is that the derivative $f'(x)$ gives us a search direction for convex functions. In particular, let $x^* := \operatorname{argmin}_{x \in [0, 1]} f(x)$, and suppose that for some $x \in [0, 1]$, $f'(x) > 0$. Then we claim that the minimizer x^* cannot lie right of x : indeed,

$$f(y) \geq f(x) + f'(x)(y - x) > f(x) \text{ for all } y > x.$$

The first inequality was (12), and the second used $f'(x) > 0$. Analogously, if $f'(x) < 0$, we can conclude that the minimizer x^* cannot lie left of x . This implies that if we can query the derivative $f'(x)$, we can perform binary search to quickly narrow an interval containing x^* .

Finally, we mention without proof a quick way to verify convexity.

Fact 1. *If $f : \mathbb{R} \rightarrow \mathbb{R}$ is twice-differentiable, then f is convex iff $f''(x) \geq 0$ for all $x \in \mathbb{R}$.*

Fact 1 follows by using a similar idea to (13) again. It implies that the gradient f' of a convex f is an increasing function. This matches our intuition when, e.g., $f(x) = ax^2 + bx + c$ is a quadratic: here, $f'(x) = 2ax + b$, which is increasing iff $a \geq 0$, which is also when f is convex.

Smoothness. In 1-d, the ability to query the gradient is already enough to employ binary search to find the minimizer. However, in high dimensions, our strategy must change, because there are infinitely many search directions, so binary search is not well-defined.

²We will hence stop bolding decision variables, i.e., a vector $\mathbf{x}$, as all variables are scalars.

This is where GD comes into play: rather than use the sign of $f'(x)$ to eliminate a portion of our search space, we instead use it to gradually guide an iterate x towards x^* . More concretely, for a *step size* $\eta > 0$, GD repeatedly iterates:

$$x \leftarrow x - \eta f'(x). \quad (14)$$

As a sanity check, for any $\eta > 0$, the update proceeds in the correct direction: if $f'(x) > 0$, then (14) moves x leftwards, where the minimizer x^* must lie. However, we can get into trouble if (14) overshoots x^* , i.e., η is too large, in which case we must backtrack. On the other hand, if we set η too small, then we can make very little progress, leading to a slow algorithm.

Why does (14) scale our movement proportionally to $f'(x)$? One reason is that the further away from the minimizer x^* we are, the larger $|f'(x)|$ is. This is a consequence of the fact that $f'' \geq 0$ pointwise, e.g., $f'(x^*) = 0$, and as we increase x from x^* , $f'(x)$ grows.

What η should we choose in (14)? A common structural property that can be used to prescribe a step size is *smoothness*. Informally, smoothness means that f' changes slowly. The intuition is that if f' is stable, then it continues to be a good descent direction for some time, and hence we do not need to worry about overshooting the minimizer for an appropriate η .

To make this intuition quantitative, we introduce two more definitions.

Definition 2 (Lipschitzness). *For $\mathcal{X} \subseteq \mathbb{R}$ and $L > 0$, we say that $f : \mathcal{X} \rightarrow \mathbb{R}$ is L -Lipschitz if*

$$|f(x) - f(y)| \leq L|x - y| \text{ for all } x, y \in \mathcal{X}.$$

Definition 2 says that if x, y are nearby points, $f(x), f(y)$ are also close. This condition can be used to argue that if we have localized the minimizer x^* in a small interval, then all points in that interval are approximate minimizers, because their function values are close to $f(x^*)$.

Finally, we define smoothness, i.e., the property of having a slowly-changing gradient.

Definition 3 (Smoothness). *For $\mathcal{X} \subseteq \mathbb{R}$ and $L > 0$, we say that differentiable $f : \mathcal{X} \rightarrow \mathbb{R}$ is L -smooth if its derivative f' is L -Lipschitz.*

Gradient access. Why are we allowed to take the gradient descent step (14) at all? In particular, how do we compute $f'(x)$? Although we will not prove this, it turns out that the intuition (3) can be quantified for smooth functions, such that the error is negligible if δ is chosen, say, at machine precision level. Thus, under smoothness it is reasonable to assume that we can access the gradient $f'(x)$, using just two function evaluations, using the formula (3).

3.2 Quadratics

The canonical example of a smooth, convex function is a *quadratic*, i.e., $q(x) = ax^2 + bx + c$. Analyzing the behavior of GD on quadratics is an important first step towards understanding its general behavior on smooth functions, so we begin by characterizing this simple case.

We assume that $a > 0$. Observe that by Fact 1 all quadratics q with $a > 0$ are convex: $q''(x) = 2a > 0$. Further, recall that to minimize a quadratic, we can complete the square:

$$\begin{aligned} x^* &:= \operatorname{argmin}_{x \in \mathbb{R}} \{q(x)\} = \operatorname{argmin}_{x \in \mathbb{R}} \{ax^2 + bx + c\} \\ &= \operatorname{argmin}_{x \in \mathbb{R}} \left\{ a \left(x + \frac{b}{2a} \right)^2 + c - \left(\frac{b}{2a} \right)^2 \right\} \\ &= \operatorname{argmin}_{x \in \mathbb{R}} \left\{ a \left(x + \frac{b}{2a} \right)^2 \right\} = -\frac{b}{2a}. \end{aligned} \quad (15)$$

Note also that the gradient of a quadratic has a clean interpretation, in light of (15):

$$q'(x) = 2ax + b = 2a(x - x^*). \quad (16)$$

The gradient (16) is thus directly proportional to the distance from our current point x to the true minimizer, x^* . Moreover, it is straightforward to check that q is $L = 2a$ -smooth:

$$|q'(x) - q'(y)| = 2a|(x - x^*) - (y - x^*)| = 2a|x - y|.$$

Therefore, for quadratic functions, it is optimal to choose the step size $\eta = \frac{1}{L}$ in the gradient descent update (14), because such a step size will take us to the minimizer in a single step:

$$x - \frac{1}{L}q'(x) = x - \frac{1}{2a}(2a(x - x^*)) = x - (x - x^*) = x^*. \quad (17)$$

More generally, choosing a step size $\eta < \frac{1}{L}$ will cause us to stay on the current side of x^* , whereas choosing $\eta > \frac{1}{L}$ overshoots the target. We can thus view $\eta = \frac{1}{L}$ as a “sweet spot” for quadratics.

Verifying Lipschitzness and smoothness. The fact that q is $2a$ -smooth, and $q'' = 2a$, is no coincidence. More generally, if f is L -smooth and twice-differentiable, taking limits shows that

$$f''(x) = \left| \lim_{y \rightarrow x} \frac{f'(y) - f'(x)}{y - x} \right| = \lim_{y \rightarrow x} \frac{|f'(y) - f'(x)|}{|y - x|} \leq L. \quad (18)$$

It turns out that the converse is also true: if $|f''(x)| \leq L$ for all x , then f is L -smooth.

Completely analogously, if f is L -Lipschitz and differentiable, then we have

$$|f'(x)| = \left| \lim_{y \rightarrow x} \frac{f(y) - f(x)}{y - x} \right| = \lim_{y \rightarrow x} \frac{|f(y) - f(x)|}{|y - x|} \leq L,$$

and the converse ($|f'(x)| \leq L$ for all x implies f is L -Lipschitz) also holds.

GD progress on quadratics. Finally, we mention one additional interpretation of the progress of gradient descent on quadratics, that extends to more general functions. In particular, we can rewrite the amount of function progress made by the update $x \leftarrow x - \frac{1}{L}q'(x)$ as follows:

$$q(x) - q\left(x - \frac{1}{L}q'(x)\right) = q(x) - q(x^*) = a(x - x^*)^2 = \frac{1}{2L}(q'(x))^2. \quad (19)$$

The first equality used that for quadratics, (17) holds, the second equality used our earlier completion of the square in (16), and the last equality used the gradient calculation in (16). Thus, the amount of progress made by the optimal gradient step is proportional to the squared gradient size. This is intuitive, as larger gradients mean there is more progress to be made.

3.3 Smooth functions

In fact, the step size $\eta = \frac{1}{L}$ is an excellent choice for smooth functions beyond quadratics.³ The main reason is that smooth functions can be upper bounded by quadratics. This lets us use the analysis in Section 3.2 as a proxy for the progress of gradient descent.

More specifically, just as the Euler approximation in (13) uses a linear function to approximate f , we can obtain sharper and sharper characterizations using higher-degree polynomials. For example, we can approximate f with a degree-2 polynomial, i.e., a quadratic:

$$f(y) \approx f(x) + f'(x)(y - x) + \frac{1}{2}f''(x)(y - x)^2. \quad (20)$$

The reason there is a coefficient of $\frac{1}{2}$ in front of the quadratic term is so that the two sides match when f is exactly a quadratic. Indeed, if we take two derivatives of both sides at $y = x$, we indeed obtain $f''(x) = f''(x)$, because the second derivative of $\frac{1}{2}f''(x)(y - x)^2$ is $f''(x)$.

Intuitively, because $f''(x) \leq L$ pointwise, we should expect that in light of (20), the upper bound

$$f(y) \leq f(x) + f'(x)(y - x) + \frac{L}{2}(y - x)^2 \text{ for all } x, y \in \mathbb{R} \quad (21)$$

holds. With a bit of calculus, we can show that this is the case even if f is not twice-differentiable.

Lemma 2. *Let $L > 0$, and let $f : \mathbb{R} \rightarrow \mathbb{R}$ be L -smooth. Then (21) holds.*

³Relatedly to (10), one smooth convex function which sees significant use in machine learning is $f(x) = \log(1 + \exp(x))$. Its derivative is always in $[0, 1]$, so it can be interpreted as a probability in binary classification, e.g., if we want to label an image, it can represent our estimated likelihood the image is a dog rather than a cat.

Proof. By the fundamental theorem of calculus,

$$f(y) = f(x) + \int_x^y f'(z) dz.$$

However, we also have $f'(z) - f'(x) \leq L|z - x|$ by smoothness. Thus, for $y \geq x$,

$$\begin{aligned} f(y) &\leq f(x) + \int_x^y f'(x) dz + L \int_x^y (z - x) dz \\ &= f(x) + f'(x)(y - x) + L \int_x^y (z - x) dz \\ &= f(x) + f'(x)(y - x) + \frac{L}{2} (z - x)^2 \Big|_x^y = f(x) + f'(x)(y - x) + \frac{L}{2} (y - x)^2. \end{aligned}$$

This proves the claim (21) for $y \geq x$. To prove (21) for $y < x$, the same proof applies after changing the first line to use $f'(z) \geq f'(x) - L|z - x|$ for $z < x$. $\square$

In summary, (12) and (21) show that for smooth, convex functions,

$$f(x) + f'(x)(y - x) \leq f(y) \leq f(x) + f'(x)(y - x) + \frac{L}{2} (y - x)^2 \text{ for all } x, y \in \mathbb{R}.$$

In other words, for any choice of centerpoint x , we can extrapolate a linear function and a quadratic function from x that sandwich the function f everywhere.

Critical points. One reasonable termination criterion for an optimization algorithm is to find a point where the gradient is small. We call a point where $|f'(x)| \leq \epsilon$ an ϵ -critical point. This definition is motivated by the fact for smooth f , $f'(x^*) = 0$ at the minimizer x^* , so it is reasonable to expect that points with small gradients lie close to x^* (though formally, only the reverse implication follows from smoothness, because we can only bound gradients by distances, not vice versa).

There is a simple argument showing gradient descent quickly returns an approximate critical point.

Lemma 3. *Let $L, \Delta, \epsilon > 0$, let $f : \mathbb{R} \rightarrow \mathbb{R}$ be L -smooth, and let $x^* := \operatorname{argmin}_{x \in \mathbb{R}} f(x)$. Let $x_0 \in \mathbb{R}$ satisfy $f(x_0) - f(x^*) \leq \Delta$. Then for $T \geq \frac{2L\Delta}{\epsilon^2}$, iterating the update*

$$x_t \leftarrow x_{t-1} - \frac{1}{L} f'(x_{t-1}), \text{ for all } t \in [T],$$

returns iterates $\{x_t\}_{t=0}^T$ such that for at least one iterate, $|f'(x_t)| \leq \epsilon$.

Proof. Fix an iteration $t \in [T]$, and let $x := x_{t-1}$ for now. Let $q(y) := f(x) + f'(x)(y - x) + \frac{L}{2} (y - x)^2$ be the quadratic upper bound in (21), such that $q(y) \geq f(y)$ for all $y \in \mathbb{R}$. We make two observations regarding q . First, $q'(y) = f'(x) + L(y - x)$, so evaluating this at x , we have $q'(x) = f'(x)$. Second, q is a quadratic with smoothness $L = q''$, so it is minimized by $x \leftarrow x - \frac{1}{L} q'(x)$.

Using these facts in conjunction with our earlier calculation (19), we have

$$f\left(x - \frac{1}{L} f'(x)\right) \leq q\left(x - \frac{1}{L} q'(x)\right) = q(x) - \frac{1}{2L} (q'(x))^2 = f(x) - \frac{1}{2L} (f'(x))^2. \quad (22)$$

In particular, this shows that for all $t \in [T]$,

$$f(x_t) \leq f(x_{t-1}) - \frac{1}{2L} (f'(x_{t-1}))^2.$$

Now suppose it were the case that $|f'(x_t)| > \epsilon$ in every iteration $0 \leq t \leq T$. Then, we have

$$f(x^*) \leq f(x_T) \leq f(x_{T-1}) - \frac{\epsilon^2}{2L} \leq f(x_{T-2}) - \frac{2\epsilon^2}{2L} \leq \dots \leq f(x_0) - \frac{T\epsilon^2}{2L}.$$

However, this would be a contradiction for $T \geq \frac{2L\Delta}{\epsilon^2}$, because we assumed $f(x_0) - f(x^*) \leq \Delta$. $\square$

Another way of phrasing Lemma 3 is that after T iterations, we will find a point x_t with

$$|f'(x_t)| \leq \sqrt{\frac{2L\Delta}{T}}. \quad (23)$$

Remarkably, Lemma 3 never used convexity. This shows GD is a reasonable strategy for finding approximate *local minima* of smooth, but potentially nonconvex, functions. This fact is often cited as an explanation for the empirical performance of GD for training neural networks, whose associated loss functions are generally very complicated, nonconvex functions.

Global minimization. If we additionally have convexity, we can obtain much stronger global optimality guarantees. We require one helper lemma whose proof can be skipped on a first read.

Lemma 4. *Let $L > 0$, let $f : \mathbb{R} \rightarrow \mathbb{R}$ be L -smooth and convex, and let $x^* := \operatorname{argmin}_{x \in \mathbb{R}} f(x)$. Then,*

$$\left| \left(x - \frac{1}{L} f'(x) \right) - x^* \right| \leq |x - x^*|, \text{ for all } x \in \mathbb{R}.$$

Proof. By squaring both sides and canceling terms, we see that it suffices to prove

$$\begin{aligned} (x - x^*)^2 - \frac{2}{L} f'(x)(x - x^*) + \frac{1}{L^2} (f'(x))^2 &\leq (x - x^*)^2 \\ \iff \frac{1}{2L} (f'(x))^2 &\leq f'(x)(x - x^*). \end{aligned}$$

However, the latter inequality follows from

$$f'(x)(x - x^*) \geq f(x) - f(x^*) \geq f(x) - f\left(x - \frac{1}{L} f'(x)\right) \geq \frac{1}{2L} (f'(x))^2.$$

Here, the first inequality used the convexity lower bound $f(x) + f'(x)(x^* - x) \leq f(x^*)$, the second used that f is minimized at x^* , and the last used our earlier calculation (22). $\square$

Lemma 4 says steps (14) with $\eta = \frac{1}{L}$ never drift further away from the minimizer. With this fact, we can prove that GD efficiently *globally minimizes* smooth, convex functions.

Lemma 5. *Let $L > 0$, let $f : \mathbb{R} \rightarrow \mathbb{R}$ be L -smooth and convex, and let $x^* := \operatorname{argmin}_{x \in \mathbb{R}} f(x)$. Let $x_0 \in \mathbb{R}$ satisfy $|x_0 - x^*| \leq R$, and let $\epsilon > 0$. Then for $T \geq \frac{L^2 R^4}{\epsilon^2}$, iterating the update*

$$x_t \leftarrow x_{t-1} - \frac{1}{L} f'(x_{t-1}), \text{ for all } t \in [T]$$

returns iterates $\{x_t\}_{t=0}^T$ such that for at least one iterate, $f(x_t) \leq f(x^*) + \epsilon$.

Proof. We first claim that it suffices to choose $\Delta = \frac{LR^2}{2}$ in Lemma 3. Indeed, because $f'(x^*) = 0$, the upper bound (21) applied at $x \leftarrow x^*$ and $y \leftarrow x$ shows that

$$f(x_0) \leq f(x^*) + \frac{L}{2} (x_0 - x^*)^2 \leq f(x^*) + \frac{LR^2}{2}.$$

Hence, after T iterations, Lemma 3 (in particular, (23)) implies we have an iterate x_t satisfying

$$|f'(x_t)| \leq \sqrt{\frac{2L\Delta}{T}} \leq \sqrt{2 \cdot \frac{L^2 R^2}{2} \cdot \frac{\epsilon^2}{L^2 R^4}} = \frac{\epsilon}{R}.$$

We conclude by applying convexity, which shows

$$f(x_t) - f(x^*) \leq f'(x_t)(x_t - x^*) \leq |f'(x_t)| |x_t - x^*| \leq \epsilon.$$

In the last inequality, we used that repeatedly applying Lemma 4 yields $|x_t - x^*| \leq |x_0 - x^*| \leq R$. $\square$

We remark that because GD makes monotone function progress, i.e., $f(x_t) \leq f(x_{t-1})$ for each iteration t (due to (22)), we can upgrade Lemma 5 to read $f(x_T) \leq f(x^*) + \epsilon$.

3.4 ...and beyond

We briefly overview improvements and generalizations of our gradient descent analysis.

Improvements. A tighter analysis of the exact same gradient descent algorithm in Lemma 5 establishes that only $T \geq \frac{LR^2}{\epsilon}$ iterations are needed to obtain an ϵ -approximate minimizer. We refer the reader to Theorem 3, [Tia24b], for a proof of this quadratic factor improvement.

By changing the algorithm, it turns out that it is possible to achieve yet another quadratic factor improvement: $T \geq (\frac{LR^2}{\epsilon})^{1/2}$ iterations suffice for smooth convex optimization [Nes83]. The key idea behind this modified algorithm is the concept of *momentum*: updating using a weighted average of gradients encountered. Intuitively, smoothness causes gradients to change slowly, so we can use the history of our algorithm as additional information to guide our iterates. Momentum is a powerful tool even in nonconvex settings, and indeed, most optimization packages used for training modern machine learning models include a tunable momentum parameter.

High dimensions. As discussed earlier in these notes, and as you will explore in your homework, much less sophisticated strategies than gradient descent (e.g., binary and ternary search) suffice for minimizing appropriately structured one-dimensional functions. However, one appealing aspect of gradient descent is that essentially the entire analysis in Section 3.3 extends to high dimensions. Concretely, if $\mathbf{x} \in \mathbb{R}^d$ and $f : \mathbb{R}^d \rightarrow \mathbb{R}$, we can define the high-dimensional gradient

$$\nabla f(\mathbf{x}) := \begin{pmatrix} \frac{\partial f}{\partial \mathbf{x}_1}(\mathbf{x}) & \frac{\partial f}{\partial \mathbf{x}_2}(\mathbf{x}) & \dots & \frac{\partial f}{\partial \mathbf{x}_d}(\mathbf{x}) \end{pmatrix} \in \mathbb{R}^d. \quad (24)$$

This direction can then guide our iterate, via the update $\mathbf{x} \leftarrow \mathbf{x} - \frac{1}{L} \nabla f(\mathbf{x})$. For L -smooth, convex $f : \mathbb{R}^d \rightarrow \mathbb{R}$, minimized at $\mathbf{x}^*$, initializing gradient descent (with momentum) at $\mathbf{x}_0 \in \mathbb{R}^d$ satisfying $\|\mathbf{x}_0 - \mathbf{x}^*\|_2 \leq R$ implies $T \geq (\frac{LR^2}{\epsilon})^{1/2}$ iterations again yields an ϵ -approximate minimizer. Analogs of Lemma 3 for finding critical points of nonconvex functions also hold.

That the quantity T does not grow in the dimension d has been enormously useful in machine learning, where decision variables (corresponding to the parameters of a deep neural network) are extremely high-dimensional, e.g., ChatGPT is estimated to have $d \approx 10^{12}$.

Further reading

For more on Section 2, see Chapters 2, 3, 6, 7, and 12, [HR22].

For more on Section 3, see Chapter 5, [HR22].

References

- [Elm90] Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990.
- [HJA20] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. In *Advances in Neural Information Processing Systems*, volume 33, 2020.
- [HR22] Moritz Hardt and Benjamin Recht. *Patterns, Predictions, and Actions: Foundations of Machine Learning*. Princeton University Press, 2022.
- [HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.
- [HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.
- [KB15] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015*, 2015.
- [KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 2012.
- [LBBH98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.
- [LH19] Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *7th International Conference on Learning Representations, ICLR 2019*, 2019.
- [Nes83] Yurii Nesterov. A method for solving a convex programming problem with convergence rate $o(1/k^2)$. *Doklady AN SSSR*, 269:543–547, 1983.
- [OWJ⁺22] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul F. Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022*, 2022.
- [SSDK⁺21] Yang Song, Jascha Sohl-Dickstein, Diederik P. Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations. In *International Conference on Learning Representations*, 2021.
- [SWD⁺17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.
- [Tia24a] Kevin Tian. Cs 395t: Continuous algorithms. <https://kjtian.github.io/cs395t.html>, 2024.
- [Tia24b] Kevin Tian. Cs 395t: Continuous algorithms, part ii (gradient descent). [https://kjtian.github.io/notes/CS%20395T%20\(Spring%202024\)/Part2_main.pdf](https://kjtian.github.io/notes/CS%20395T%20(Spring%202024)/Part2_main.pdf), 2024.
- [VSP⁺17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 30, 2017.