Gradient descent is a very different algorithm for solving linear equations, and more. It is the main algorithm responsible for training machine learning models.

Gradient descent is an optimization algorithm. The goal of optimization is to find the best possible solution among many. How do we phrase solving linear equations as an optimization problem? The idea is to set up some "loss function" f that penalizes guesses which are far from being correct. The minimum of the loss function is attained only by the correct solution(s).

Let's start with a silly but enlightening example. The equation is $2x = 3$. We want to cook up a function f that is uniquely minimized at the solution point $x = 3/2$. What would be the simplest function with this property? Linear functions won't work as they get smaller towards one of the infinities. The next simplest type is quadratic and we are in luck: The function $(x - 3/2)^2$ has the solution as its unique minimizer. (Another option is the piecewise linear function $|x - 3/2|$. We'll come back to it later.)

We want to set up f in order so solve the equation $2x = 3$. But in order to write up f we had to figure out the solution $3/2$ first. This isn't as outlandish as it looks. We didn't really have to solve any equation. If we "unsolve" it we get $(2(x - 3/2))^2 = (2x - 3)^2$ which is left minus right hand side squared. This function is minimized precisely at $x = 3/2$.

To set up a loss function for a problem with multiple constraints like solving a system of equations, it is natural to add up the losses of the individual constraints. The resulting function will be minimized exactly when all constraints are satisfied.

To summarize, one loss function for a linear system is the sum of left minus right hand sides squared. This is called the *sum of squares loss*. For last lecture's example,

$$\begin{aligned} x - 3y - 3z &= -5 \\ x - 2y - 2z &= 0 \\ x + y - 5z &= 3 \end{aligned} \tag{1}$$

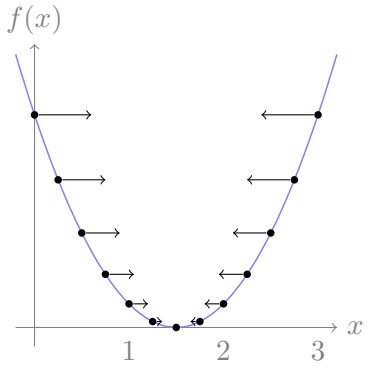
the sum of squares loss is

$$f(x, y, z) = (x - 3y - 3z + 5)^2 + (x - 2y - 2z)^2 + (x + y - 5z - 3)^2. \tag{2}$$

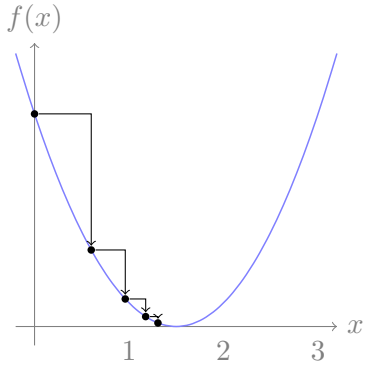
It is useful to think of (x, y, z) as the position of a particle. Gradient descent starts with this particle somewhere, say at $(0, 0, 0)$, and keeps moving it in a direction in which the sum of squares loss decreases. When it is convinced that no decrease is possible it declares that the minimum has been found. The name of the algorithm comes from direction of motion: the *gradient* of f, with a minus sign in front.

1 Gradient descent

Let's start with $f(x) = (2x-3)^2$. The gradient of a univariate function is the derivative $df/dx = 4(2x-3)$. Here is a plot of $-0.05 \cdot df(x)/dx$ at various points along the curve $(x, f(x))$:



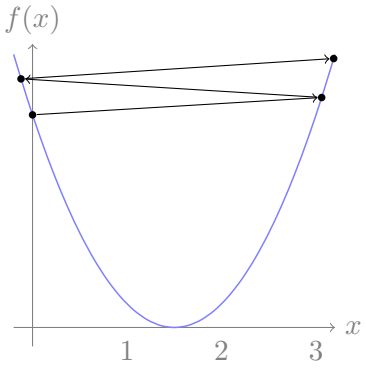
All arrows point towards the minimum. This suggest we should move x by $-0.05 \cdot df(x)/dx$ units in each step. If the particle is initially at $x = 0$ it will trace this path on the graph $(x, f(x))$, eventually reaching close to the minimum at $x = 1.5$.



This is the gradient descent algorithm. The gradient of an n-variate function $f(\mathbf{x}) = f(x, y, \ldots, z)$ is the vector of partial derivatives $\nabla f = (\partial f/\partial x, \partial f/\partial y, \ldots, \partial f/\partial z)$.

Algorithm *GD* (Simple Gradient Descent)
Input: A function f with multiple inputs.
1 Choose a starting point $\mathbf{x} = (x, y, \ldots, z)$.
2 Choose a rate $\rho > 0$.
3 Until $\mathbf{x}$ is sufficiently small,
4 Move $\mathbf{x}$ to $\mathbf{x} - \rho \cdot \nabla f(\mathbf{x})$.

Gradient Descent is different from Gaussian Elimination in many ways. The first glaring difference is that it is not "set in stone": We need to choose the initial $\mathbf{x}$ in line 1 and the rate ρ in line 2. The rate controls the speed of motion. Set it too small and the particle will take forever to budge. Set it too high and it will go in the wrong direction! Had I chosen a rate of 0.255 instead of 0.05 this is the path it would have followed:



Even if the parameters are chosen well, Gradient Descent never quite arrives at its destination. In line 3 we have to decide when to declare victory. There are guidelines that might govern these choices but it is difficult to get a feel about Gradient Descent without some trial and error. Let's see what happens when we apply it to system (1).

The first task is to calculate the gradient of the square loss. Applying the usual rules from calculus,

$$\begin{aligned}
\partial f/\partial x &= 2 \cdot (x - 3y - 3z + 5) + 2 \cdot (x - 2y - 2z) + 2(x + y - 5z - 3) \\
\partial f/\partial y &= 2 \cdot (-3)(x - 3y - 3z + 5) + 2 \cdot (-2)(x - 2y - 2z) + 2 \cdot (x + y - 5z - 3) \\
\partial f/\partial z &= 2 \cdot (-3)(x - 3y - 3z + 5) + 2 \cdot (-2)(x - 2y - 2z) + 2 \cdot (-5)(x + y - 5z - 3).
\end{aligned}$$

Each partial derivative is a linear combination of the equations, with the coefficient equal twice to the one in front of the corresponding variable. This can be written succinctly in matrix notation. If the system has the form $A\mathbf{x} = b$ and $f(\mathbf{x}) = \|A\mathbf{x} - b\|^2$ then

$$\nabla f(\mathbf{x}) = 2A^T(A\mathbf{x} - b), \tag{3}$$

where A^T is the matrix transpose of A. (Later in the course we will derive this formula as an instance of backpropagation.) Now we can start playing.

Let the olympic games begin!

We first need to initialize $\mathbf{x}$. It is reasonable to expect that the closer we start to the minimum the faster we will get there. From last week's problem formulation we know that the variables represent people's ages so they should not be negative or too large. Let's set the initial $\mathbf{x}$ to $(0, 0, 0)$.

How about the rate? Let's try to get a sense of scale. The rate is the multiple of the gradient by which the particle moves. The solution should be single or double-digit numbers, which suggests that the rate should be roughly inverse-proportional to the norm (the length) of the gradient. We can get a rough estimate by evaluating the gradient at the initial point. In our example, formula (3) gives $\nabla f(0, 0, 0) = 2A^T b = (-4, 36, 0)^T$, which has norm about 36. This suggests $\rho \approx 1/36 \approx 0.03$.

A few trial runs reveal that gradient descent diverges on system (1) with $\rho = 0.03$, but seems to make progress when ρ is set to 0.02. Here are the first ten steps:

```
step   x                        root(f(x))
1    : [0.      0.     0.   ]  5.831
2    : [-0.08   0.72   0.   ]  3.937
3    : [-0.035  1.024 -0.262]  3.178
4    : [-0.052  1.249 -0.205]  2.912
5    : [-0.008  1.327 -0.314]  2.822
6    : [-0.004  1.403 -0.265]  2.788
7    : [ 0.038  1.422 -0.311]  2.771
```

```
8    :  [ 0.057  1.451 -0.278]  2.760
9    :  [ 0.091  1.457 -0.297]  2.751
10   :  [ 0.114  1.471 -0.275]  2.742
```

The particle moves rapidly in the first two steps but then progress slows down. It takes a long time for it to get close to the target solution $(10, 3, 2)$.

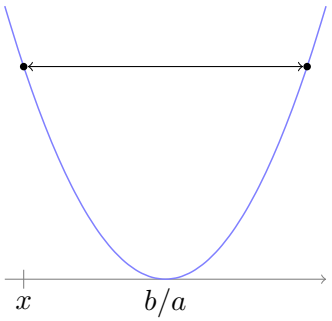
```
step   x                        root(f(x))
200 :  [4.26    2.113  0.674]   1.592
400 :  [6.761   2.5    1.252]   0.898
600 :  [8.172   2.718  1.578]   0.507
800 :  [8.969   2.841  1.762]   0.286
1000:  [9.418   2.91   1.866]   0.161
2000:  [9.994   2.999  1.999]   0.002
```

Gradient descent looks very fickle, and takes thousands of steps to complete what an average sixth-grader can do in five minutes. Is it worth the trouble? Absolutely!

2 Gradient Descent fast and slow

Analyzing gradient descent is tricky. Fortunately we can learn quite a bit by looking at the one-variable equation $ax = b$. To visualize more easily let's assume that a and b are positive.

The most pressing question is why Gradient Descent works at all. The proof by picture from Section 1 should be convincing enough but let's try to quantify which rates ρ result in convergence. There is some precise value ρ^* at which the particle keeps bouncing between two points on the curve:



We can find ρ^* by solving an equation. The displacement between these two points is $2(b/a - x)$. Gradient Descent moves x by $-\rho \cdot df(x)/dx = -\rho \cdot 2a(ax - b)$, so ρ^* must be the value of ρ at which these two expressions are equal, namely

$$\rho^* = \frac{2(b/a - x)}{-2a(ax - b)} = \frac{1}{a^2}.$$

As long as $\rho < 1/a^2$, Gradient Descent will make progress.

To say something similar about general systems we need a single number that measures the "magnitude" of the matrix A. This is the *spectral norm* $\|A\|$. As long as $\rho < 1/\|A\|^2$, Gradient Descent is guaranteed to make progress. We'll define and explain the spectral norm shortly.

The next question is how fast this convergence is. How much closer to the minimum does each step take us? If the particle moves from x to x', then

$$x' = x - \rho \cdot 2a(ax - b).$$

This equation homogenizes to $y' = (1 - 2a^2\rho)y$ for $y = ax - b$, $y' = ax' - b$. The values y and y' are the square roots of the loss before and after a step of Gradient Descent. In each step this root-loss shrinks

by a factor of $(1 - 2a^2\rho)$, that is at an exponential rate as long as ρ is bounded away from 0 and $1/a^2$. Exponential convergence rates are very desirable. They mean that every few steps we get closer to the solution by an extra digit of accuracy.

The picture is more complicated for more equations and more unknowns. Assuming $\rho < 1/\|A\|^2$, if our initial guess is at distance d from some solution to $A\mathbf{x} = b$, it is known that

$$f(\mathbf{x}_t) = \text{sum of squares loss after } t \text{ steps} \leq \frac{d^2}{(1 - \rho\|A\|^2)\rho t}. \tag{4}$$

The upper bound on the right guarantees progress, but at a merely inverse linear rate $1/t$. (The rate becomes worse as ρ approaches the boundaries 0 and $1/\|A\|^2$.) This is a far cry from the exponential convergence in univariate systems.

How does this square with our experience so far? In example (1), $\|A\|$ is about 6.58. This suggests choosing a rate ρ between 0 and $1/6.58^2 \approx 0.023$. Our choice $\rho = 0.02$ just about fits the requirement. The squared distance d^2 between our initial guess $(0, 0, 0)$ and the solution $(10, 3, 2)$ is $10^2 + 3^2 + 2^2 = 113$. Formula (4) guarantees sum of squares loss rate at most about $42053/t$. It is more natural to look at its square root which is the norm of the error vector. It should be at most about $205/\sqrt{t}$. Let's compare this upper bound to our experiment.

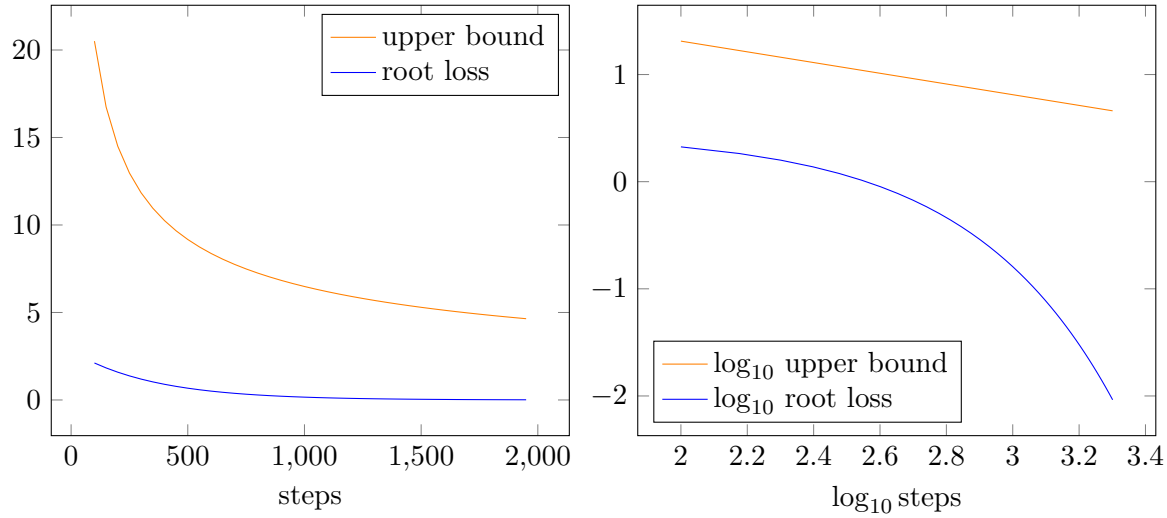


FIGURE 1: Loss as a function of time at (a) regular (b) log-log scale.

In the left plot x-axis is the step t, the blue line is the root loss $\sqrt{f(\mathbf{x}_t)}$, and the orange line is the root of the upper bound on the right-hand side of (4). The algorithm does quite a bit better than the bound. (I am not sure why; this could be another good project.) Both curves have a similar shape. To get more detail, the right plot displays the same data at log-scale. The upper bound in (4) is on the order of $1/\sqrt{t} = t^{-1/2}$. The exponent $-1/2$ is the slope of the orange line on the right. The blue line has the same slope initially, suggesting convergence at rate $1/\sqrt{t}$, but takes a plunge around step $10^{2.6} \approx 400$. Why does this happen?

3 The spectral norm and the condition number

To understand this phenomenon we must look beyond one equation in one unknown. The next simplest type of instance is two equations in two unknowns. Take a look at these two systems:

$$S\colon \begin{aligned} x + 0.9 \cdot y &= 1 \\ x + 1.1 \cdot y &= 1 \end{aligned} \qquad \text{and} \qquad T\colon \begin{aligned} 0.9 \cdot x + y &= 1 \\ 1.1 \cdot x + y &= 1 \end{aligned}$$

Systems S and T look very close. Their right-hand sides are identical, and all the coefficients are close. Their solutions, however, are dramatically different: S has $x = 1, y = 0$ as its unique solution, while in T the roles are reversed: $x = 0$ and $y = 1$. If two systems of equations are "close" it doesn't mean that so must be their solutions. Yet Gradient Descent (unlike Gaussian Elimination again) is a "smooth" algorithm: If we slightly perturb the instance, it should take many steps for the particle to notice. When does this happen?

Suppose our initial guess for a solution in S is $(1/2, 1/2)$. The actual solution is $\mathbf{x}^* = (1, 0)$. We can write a matrix-vector recurrence like in the one-variable case. The gradient descent update rule is

$$\mathbf{x}' = \mathbf{x} - \rho\nabla f(\mathbf{x}) = \mathbf{x} - 2\rho A^T(A\mathbf{x} - b).$$

Substituting $b = A\mathbf{x}^*$ and subtracting another $\mathbf{x}^*$ from both sides gives

$$\mathbf{x}' - \mathbf{x}^* = (\mathbf{x} - \mathbf{x}^*) - 2\rho A^T A(\mathbf{x} - \mathbf{x}^*)$$

Using I to denote the identity matrix, we can factor the right-hand side to obtain vector recurrence

$$\mathbf{x}' - \mathbf{x}^* = (I - 2\rho A^T A)(\mathbf{x} - \mathbf{x}^*).$$

In each step, gradient descent moves $\mathbf{x}$ by a "factor" of $I - 2\rho A^T A$ towards the solution $\mathbf{x}^*$. The spectral norm of this factor governs the rate of convergence.

This is a good time to talk about spectral norm. The spectral norm of a matrix B is the maximum amount by which it can stretch a vector, namely

$$\|B\| = \max_{\mathbf{v}\neq 0} \frac{\|B\mathbf{v}\|}{\|\mathbf{v}\|}.$$

If the spectral norm is less than one then B shrinks every vector. If it is at most 0.9 then every vector shrinks by a factor of 0.9. If we apply B again it shrinks $0.9^2 = 0.81$ times. If we apply B a hundred times it shrinks $0.9^{100} \approx 0.00002$ times, a tiny number.

Let's write down the matrix $B = I - 2\rho A^T A$ for the system S:

$$A = \begin{bmatrix} 1 & 0.9 \\ 1 & 1.1 \end{bmatrix} \quad \longrightarrow \quad B = \begin{bmatrix} 1 - 4\rho & -4\rho \\ -4\rho & 1 - 4.04\rho \end{bmatrix}.$$

The spectral norm of B is about $1 - 0.02\rho$. We will learn how to calculate it next time. Initially, $\mathbf{x} - \mathbf{x}^* = (1/2, 1/2) - (1, 0) = (-1/2, 1/2)$ which has norm about 0.707. After t steps, $\mathbf{x}$ and $\mathbf{x}^*$ are $(1-0.02\rho)^t \cdot 0.707$-close. If we set $\rho = 0.2$, an acceptable rate (less than $1/\|A\|^2 \approx 0.25$), $\mathbf{x}$ and $\mathbf{x}^*$ would be guaranteed to be within 0.1 after 500 steps. Indeed $\mathbf{x}_{500} = (0.932, 0.068)$, which is 0.096-close to $(1, 0)$. The spectral norm of B looks like an excellent predictor of convergence speed.

Where did the factor of 0.02 come from? Here is a "pattern-matching" explanation. The spectral norm of $B = I - 2\rho A^T A$ is about $1 - 0.02\rho = 1 - 2\rho \cdot 0.01$. There is a 2ρ in both expressions, the 1 is comes from the identity matrix, so 0.01 must be the contribution of A.

The matrix A is close to the all-ones matrix. If A were the all-ones matrix, its rows would be linearly dependent, there would be many solutions, and it would be hopeless to expect convergence to any particular $\mathbf{x}^*$. The only possible explanation is that the contribution of the all-ones matrix is zero. Any matrix with linearly dependent rows contributes zero. The *the condition number* of a matrix measures how close it is to having linearly dependent rows. You will investigate it in Homework 1. Here is a summary of all this nonsense.

Theorem 1. *If A is a square matrix with linearly independent rows, $\rho < 1/\|A\|^2$, after t steps with initial guess $\mathbf{x}$, the state $\mathbf{x}_t$ of gradient descent satisfies*

$$\|\mathbf{x}_t - \mathbf{x}^*\| \leq \left|1 - 2\rho \cdot \|A\|^2\kappa^{-1}\right|^t \cdot \|\mathbf{x} - \mathbf{x}^*\|,$$

where $\mathbf{x}^*$ is the unique solution to $A\mathbf{x}^* = b$, and κ is the condition number of $A^T A$.

The key feature of this theorem is that gradient descent closes in on the solution at a rate inverse *exponential* in the number of steps t. This exponential accounts for the drooping shape on the curve in Figure 1. (Assuming a rate of about $1/\|A\|^2$, the condition number controls the base of this exponent.) In contrast, (4) only guarantees square loss rate that is inverse linear in t.

If Theorem 1 is so much better than bound (4) why did we bother with (4) at all? Actually the two bounds are hardly comparable. Equation (4) bounds the rate of decrease of the sum of squares loss $f(\mathbf{x})$, but it tells us little about how good the solution $\mathbf{x}$ is. For example, after running gradient descent on S for 100 steps (with the above parameters), $\sqrt{f(\mathbf{x}_{100})}$ is 0.048. This looks great! But the corresponding solution is $\mathbf{x}_{100} = (0.664, 0.334)$, which is a long way from $\mathbf{x}^* = (1, 0)$.

On the other hand, if the system happens to be underdetermined—say we have 10 unknowns but only 7 equations—then Theorem 1 does not apply because $\mathbf{x}^*$ cannot be uniquely defined. Bound (4), however, is relevant. It tells us that eventually the system will be almost solved by gradient descent. It doesn't say which of the many candidate solutions the algorithm picks. This is not a bug but an important feature of gradient descent. By adding extra terms to the "objective" f we can guide gradient descent to favor some solutions over others, for instance ones that tend to be short. We'll explain why this is useful in Section 6.

4 Linear regression

Gaussian Elimination struggled mightily with data analysis. This is where Gradient Descent shines. Let's try it to infer grade weights from student marks on the same data:

	h	m	p	f
Alice	87.78	96.67	80.00	87.33
Bob	100.83	100.00	95.00	98.25
Dave	84.44	76.67	80.00	80.33
Charlie	48.89	86.67	72.50	69.67
Eve	84.72	96.67	85.00	88.42

The initial guess is that all three components weigh equally: $\mathbf{x} = (1/3, 1/3, 1/3)$. As for the rate, the data is at the scale of hundreds, so we might expect ρ to be some small multiple of $1/100^2$. With $\rho = 0.05 \cdot (1/100)^2$ the result is

```
step   x                       root(f(x))
1000:  [0.31  0.32  0.368]     0.333
2000:  [0.305 0.31  0.385]     0.161
3000:  [0.302 0.305 0.393]     0.078
4000:  [0.301 0.302 0.397]     0.038
5000:  [0.3   0.301 0.398]     0.019
```

This is a very close approximation to the actual grade weights $(0.3, 0.3, 0.4)$! Let's do another experiment. When the data was rounded to the closest integer, Gaussian Elimination produced a very poor answer. Here is how Gradient Descent performs on the rounded data:

```
step   x                       root(f(x))
1000 :  [0.287 0.32  0.387]    0.608
2000 :  [0.279 0.305 0.413]    0.415
3000 :  [0.275 0.297 0.425]    0.355
4000 :  [0.273 0.293 0.431]    0.339
5000 :  [0.272 0.291 0.434]    0.335
```

This doesn't look bad at all. There is a reason behind it. Among all algorithms you could try, Gradient Descent is guaranteed to give the best possible approximation, the one that minimizes the sum of squares error. Theorem 1 generalizes to Theorem 2:

Theorem 2. *For any matrix A, $\rho < 1/\|A\|^2$, after t steps with initial guess $\mathbf{x}$, the state $\mathbf{x}_t$ of gradient descent satisfies*

$$\|\mathbf{x}_t - \mathbf{x}^*\| \leq (1 - 2\rho \cdot \kappa^{-1})^t \|\mathbf{x} - \mathbf{x}^*\|,$$

assuming $\mathbf{x}^*$ is the unique input that minimizes $f(\mathbf{x}^*) = \|A\mathbf{x}^* - b\|^2$.

A linear system that has more linearly independent equations than unknowns is called *overconstrained*. Taken literally such systems have no solutions. Yet solving them is a pressing need in data analysis and machine learning. The price of a house is likely to depend on the square footage, the median income in the neighborhood, and years since the last renovation. We do not expect it to be an exact linear combination of those three numbers. Nevertheless, a retail agent might still want to model

$$\text{price} = x \cdot \text{footage} + y \cdot \text{income} + z \cdot \text{years} + \text{noise}.$$

This data-dependent noise could be very large. We should expect it to be large for such a simplistic model of the property market. Nevertheless, Gradient Descent produces the model (x, y, z) that minimizes the average square noise based on available data.

If this model makes lousy predictions it is not because Gradient Descent underperformed. It is the model's fault. To improve predictions we should consider adding variables (turnover rate, distance to subway) or look into more complex, nonlinear models. A surprising discovery of modern machine learning is that "fine-tuned" variants of gradient descent are just as effective on complex nonlinear models.

5 Variants and applications

Yet another way in which Gradient Descent differs from Gaussian Elimination is that it its performance is highly dependent on the input. Input parameters like the spectral norm and the condition number greatly affect its rate of convergence. These parameters can sometimes be as hard to estimate as solving the system itself. Testing the algorithm with different initializations and rates can be helpful in figuring out what is best for a given input. A natural extension of GD picks the rate adaptively depending on the state:

Algorithm *AGD* (Adaptive Gradient Descent)
Input: A function f with multiple inputs.
1 Choose a starting point $\mathbf{x} = (x, y, \ldots, z)$.
3 Until $\mathbf{x}$ is sufficiently small,
2 Choose a rate ρ depending on $\mathbf{x}$, $\nabla f(\mathbf{x})$, $f(\mathbf{x})$, and the previous value of ρ.
4 Move $\mathbf{x}$ to $\mathbf{x} - \rho \cdot \nabla f(\mathbf{x})$.

There are several reasonable rules for picking the rate. In *steepest descent* ρ is chosen to minimize the target $\|\mathbf{x} - \rho \cdot \nabla f(\mathbf{x})\|$ among all possible choices of ρ. The advantage is that progress towards the minimum in any given step is as good as it can get. The disadvantage is that finding the best possible ρ requires time-consuming calculations. Another variant called *conjugate gradient descent* reduces the dependence on the condition number from κ^{-1} to $\kappa^{-1/2}$, and is guaranteed to terminate in n steps (assuming κ is finite).

To further improve convergence, it is sometimes possible to apply a transformation on the rows of the original system to make them less dependent and thereby decrease the condition number. The transformation has to be simple enough so that one can easily recover the solution to the original system from a solution to the transformed one. This is called *preconditioning*.

Gradient descent interacts with its input only by evaluating the gradient $2A^T(A\mathbf{x} - b) = 2A^TA\mathbf{x} - 2A^Tb$. It is effectively solving the system $S\mathbf{x} = b'$, with $S = A^TA$ and $b' = A^Tb'$. Matrices of the form A^TA are called *symmetric positive semidefinite* (PSD). There are many applications in which the input itself is already of this type. Gradient descent tends to be a lot more efficient on PSD inputs.

Unlike Gaussian Elimination, Gradient Descent is friendly to sparse matrices. Gradient calculation can be implemented in time proportional to the number of nonzero entries of the input matrix. This can be a dramatic speedup in some applications.

Fairly recently, theorists developed ingenious algorithms for an important class of positive semidefinite systems called Laplacians. Laplacian systems come up in calculating the *hitting time* it takes a random walk to reach one graph vertex from another, for example how many steps it would take you to reach my web page if you surfed the internet at random. Hitting times measure how important a given connection is in a computer network. These fast Laplacian solvers tread close to the limit of algorithmic efficiency. Their execution time is only a few factors larger than the time it takes to read the matrix, at least in theory. Their implementation and analysis uses all of the above ideas (conjugate gradients, sparsity, preconditioning), and then some.

Finally, a simple twist on Gradient Descent is quite effective in machine learning applications. Owing to the humongous number of parameters in modern machine learning models, the matrices there can be very large. Evaluating the gradient can be a significant bottleneck. In this context rows of matrices represent data records like the grades of a student. Instead of evaluating the gradient exactly we can try to approximate it using just a small subsample of the records. This variant is called *stochastic gradient descent*.

6 The unreasonable effectiveness of Gradient Descent

The spectacular success of modern machine learning probably comes from a confluence of several innovations: Data availability, extensive and cheap processing power, painstaking trial and error. Yet the one truly indispensable algorithm that powers it all is gradient descent. The latest models for object recognition, language translation, and chatbot interaction are all trained with gradient descent. How can an algorithm designed for solving $2x + 3 = 7$ be so powerful?

The reason is that a great many problems from all walks of engineering, science, and life can be turned into minimizing some objective function. It is not guaranteed that gradient descent will succeed on them but it can at least be attempted and played with. This paradigm has been especially effective in modern machine learning.

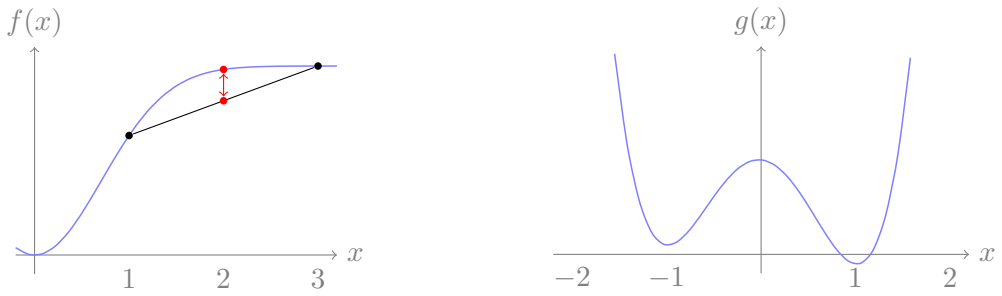


FIGURE 2: Non-convex functions. (a) $f(x) = 1 - e^{-x^2}$: The average of $f(1)$ and $f(3)$ is smaller than the value $f(2)$ at the midpoint. Gradient descent can get stuck on the flat part. (b) $g(x) = (x^2 - 1)^2 - 0.1x$: Gradient descent could converge to the local minimum $x_- \approx -1$ or to the global one $x_+ \approx 1$ depending on the choice of initialization.

For starters, the gradient descent strategies we discussed generalize almost verbatim to any *convex* objective function. A function is convex if the averaging two evaluations is never smaller than evaluating the average. For example, x^2 and $|x|$ are convex, but $1 - e^{-x^2}$ and $(x^2 - 1)^2$ are not. A consequence of convexity is that not only is there a unique minimum but the gradient always points towards it.[1]

[1]For the math nerds: If the function is not differentiable a generalized *subgradient* can be defined.

Bound (4) and Theorem 2 apply to any convex f for a suitable generalization of "spectral norm" and "condition number".

A fantastic feature of convex functions is that they can be added together. This is incredibly useful for model selection. Suppose you want to build a linear model but don't have enough data. This is the problem of *overfitting*. There are many models consistent with the data so which one should you trust? A reasonable strategy is to select the one that least complex by some criterion. If this criterion is convex, gradient descent can handle it. A common complexity measure for solutions to linear systems is the sum of magnitudes, namely $g(\mathbf{x}) = |x| + |y| + \cdots + |z|$. At the very least it rules out solutions with unusually large numbers, like one that predicts the weight of the homeworks in the final grade is 157.

One surprising more recent realization is that gradient descent is often effective even for non-convex optimization. On non-convex instances gradient descent could converge to a local minimum that is not the true, global minimum, or it might not converge at all (see Figure 2). Not only is there no guarantee that it will succeed, but it is sometimes even hard to interpret if the outcome was a success or a failure. Yet highly non-convex complex machine learning models are successfully trained using gradient descent every day. It remains a great challenge for theorists to explain why gradient descent works so far beyond the humble problem that it was designed for—minimizing sums of squares.