Lecture 11: Of balls and bowls

CS 3780/5780, Sp25

Tushaar Gangavarapu (TG352@cornell.edu)

In the previous lecture we saw *iterative* methods to finding the optimal parameters for a given cost function $J(\theta)$, starting from a reasonably good initial guess, $\theta^{(0)}$. Formally, for a given iteration G, with fixed point $\theta^\star$, we constructed iterates $\theta^{(1)}, \ldots, \theta^{(k)}$, where

$$\theta^{(k+1)} = G(\theta^{(k)}); \qquad G(\theta^\star) = \theta^\star.$$

One iteration that we built from intuition was to take "sufficiently small" steps (read: α values) in the direction of the steepest descent, given by the direction opposite to that of the gradient, until you reach the fixed point:

$$\theta^{(k+1)} = G(\theta^{(k)}) = \theta^{(k)} + \alpha(-\nabla J(\theta^{(k)})).$$

If you recall, we showed that gradient descent offers linear convergence for a strictly convex[1] quadratic landscape of the form $J(\theta) = (1/2)\theta^T A\theta + b^T\theta + c$:

$$\|\varepsilon^{(k+1)}\| = \|(\mathrm{I} - \alpha A)\varepsilon^{(k)}\|,$$

where $\varepsilon^{(k)} = \theta^\star - \theta^{(k)}$, so long as $\alpha < 2/\lambda_{\max}$, where $\lambda_{\max}$ is the largest eigenvalue of A. In our example quadratic with $A = 2\mathrm{I}, b = 0, c = 0$, which gives us $J(\theta) = \theta^T\theta = \theta_1^2 + \theta_2^2$, we noted that $\alpha < 1$ guarantees convergence.

This guarantee clearly imposes bounds on how large a step size you can take before diverging, and that depends on $\lambda_{\max}$. One could easily imagine scenarios where we are forced to set a really small α to have any chance of convergence, which is not ideal. Additionally, the cost function landscapes are not as simple as our single-minima convex objective: how do we deal with more than one minima, or, what happens when the landscape is *almost* flat in one direction but steep in the other? Finally, what are the practical (computational) considerations when implementing gradient descent? Can we do better? In this lecture, we will see specific improvements over vanilla gradient descent that allow for faster convergence, possibly jolting out of local minima, and allowing for faster computation of gradients.

1 Momentum

Let's start with our first concern: possibly slow convergence despite using an optimal α. Let us revisit our error iteration for the general quadratic to see what the fastest possible convergence rate is. We have $\|\varepsilon^{(k+1)}\| = \|(\mathrm{I}-\alpha A)\varepsilon^{(k)}\|$, and we are looking for optimal $\alpha < 2/\lambda_{\max}$. Recall that the eigenvalues of $\mathrm{I} - \alpha A$ control the amount of "stretch" it inflicts on $\varepsilon^{(k)}$. If $\lambda_{\min}$ and $\lambda_{\max}$ are the extreme eigenvalues of A, then for any setting of α, the maximum stretch caused by $\mathrm{I} - \alpha A$ must be

$$\max(|1 - \alpha\lambda_{\min}|, |1 - \alpha\lambda_{\max}|).$$

Note that above needs to be capped at 1 to ensure $\|\varepsilon^{(k+1)}\| < \|\varepsilon^{(k)}\|$, i.e., convergence. We therefore obtain the following piecewise functional form for our worst-case *convergence*:

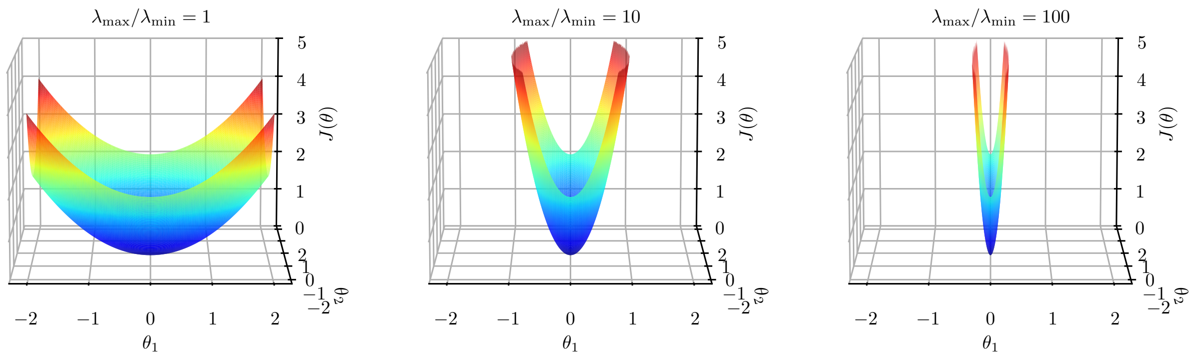
$$\begin{cases} 1 - \alpha\lambda_{\min} & 0 < \alpha < 2/(\lambda_{\min} + \lambda_{\max}) \\ \alpha\lambda_{\max} - 1; & 2/(\lambda_{\min} + \lambda_{\max}) \le \alpha < 2/\lambda_{\max}. \end{cases}$$

[1] A is symmetric and positive definite, meaning all eigenvalues are positive (see Lecture 10, §1.3: "Aside on quadratic form" for reference). Advanced: Strict and strong convexity are not interchangeable; for example $\exp(x)$ is strictly convex but not strongly convex.

Hence, setting $\alpha = 2/(\lambda_{\min} + \lambda_{\max})$ ensures neither $|1 - \alpha\lambda_{\min}|$ nor $|1 - \alpha\lambda_{\max}|$ dominate, which makes our worst-case convergence as fast as possible. How fast?

$$\max\left(\left|1 - \frac{2\lambda_{\min}}{\lambda_{\min} + \lambda_{\max}}\right|, \left|1 - \frac{2\lambda_{\max}}{\lambda_{\min} + \lambda_{\max}}\right|\right) = \frac{\lambda_{\max} - \lambda_{\min}}{\lambda_{\max} + \lambda_{\min}} = \frac{\lambda_{\max}/\lambda_{\min} - 1}{\lambda_{\max}/\lambda_{\min} + 1}.$$

Now, if A is such that $\lambda_{\min} = \lambda_{\max}$, we converge in a single step. You can verify this (manually or by running the demo[2]) by setting $\alpha = 0.5$ and run gradient descent for $J(\theta) = \theta_1^2 + \theta_2^2$. We are more interested in a non-idealistic setting, say, one where $\lambda_{\max} \gg \lambda_{\min}$—this would imply a heartbreakingly slow convergence! We observe our bowl getting narrower and steeper as the ratio, $\lambda_{\max}/\lambda_{\min}$ grows:



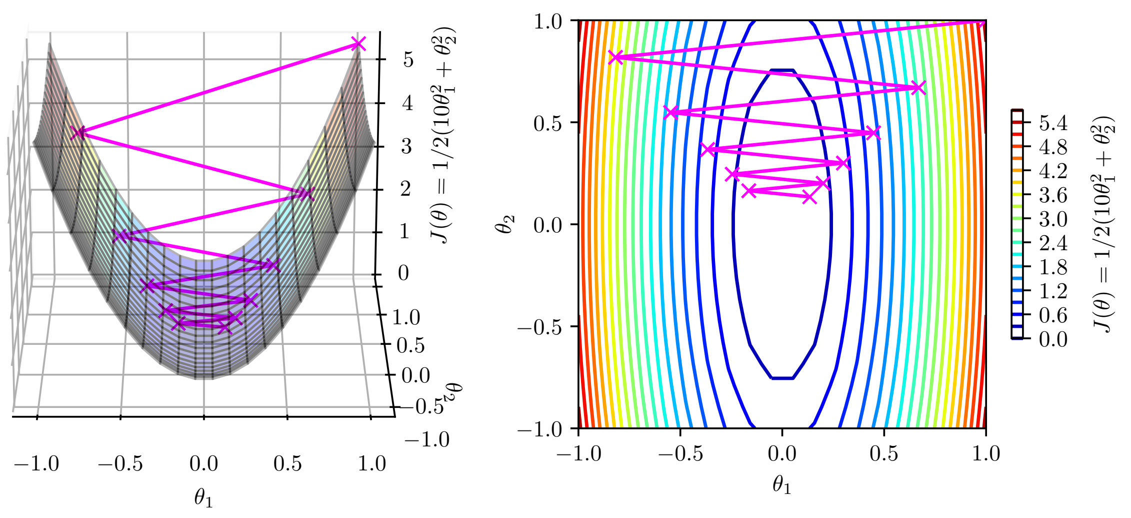
To bolster our previous point of slow convergence, let us run the gradient descent iteration when $\lambda_{\max} = 10$ and $\lambda_{\min} = 1$; we can construct such a J as:

$$J(\theta) = \frac{1}{2}\theta^T \begin{bmatrix} 10 & 0 \\ 0 & 1 \end{bmatrix} \theta = \frac{1}{2}(10\theta_1^2 + \theta_2^2); \qquad \nabla J = \begin{bmatrix} 10 & 0 \\ 0 & 1 \end{bmatrix} \theta = \begin{bmatrix} 10\theta_1 \\ \theta_2 \end{bmatrix}.$$

Starting at $\theta^{(0)} = (1, 1)$, we have the next step as

$$\theta^{(1)} = \theta^{(0)} - \alpha \nabla J = \begin{bmatrix} 1 \\ 1 \end{bmatrix} - \alpha \begin{bmatrix} 10 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 - 10\alpha \\ 1 - \alpha \end{bmatrix}.$$

From our previous discussion, we realize that the optimal α is $2/(\lambda_{\min} + \lambda_{\max}) = 2/11$, and plugging in gives us $\theta^{(1)} = (-9/11, 9/11)$. It looks like we overshot the minimum! Let's see what this looks like for a few more iterations:



[2] https://colab.research.google.com/drive/14G5hZpzHGOGr3wi_fE7hJanYsvSiiqFi.

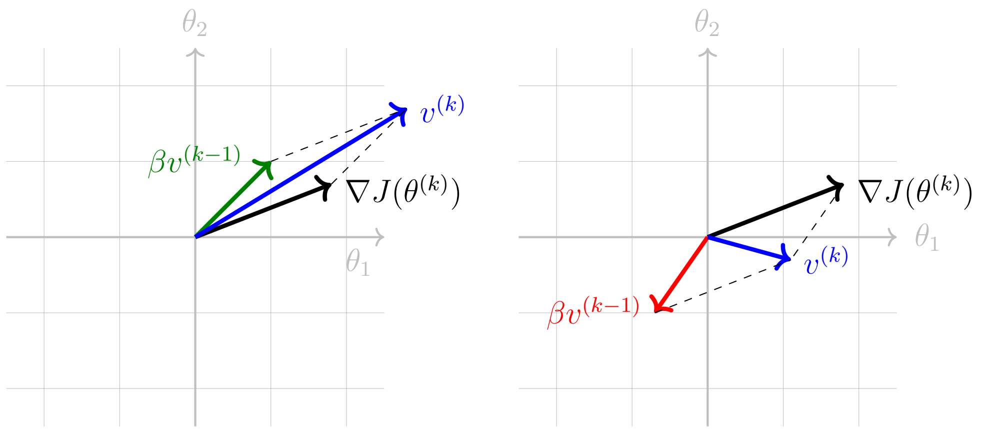
1.1 “Heavy” ball rolling downhill

The slow *zig-zag* path to convergence is a real problem with gradient descent that we need to improve. The key observation we make here is that when an external force is applied, lightweight ball will observe these zig-zag motions, i.e., jumping from one side of the bowl surface to the other. However, assuming reasonable force, this would not happen for a heavier ball rolling downhill, whose path is more stable.

What does this mean concretely?—if the previous gradient direction and the current direction are the same, we increase our velocity (similar to how a heavy ball would gain *momentum*). However, when the past and current gradients point in opposite directions, we dampen the velocity:

$$\underbrace{v^{(k)}}_{\text{velocity}} = \underbrace{\beta v^{(k-1)}}_{\substack{\text{dampened at}\\\text{each step}}} + \underbrace{\nabla J(\theta^{(k)})}_{\substack{\text{perturbed by an}\\\text{external force}}} .$$

If you’ve seen exponential smoothing before, the above is equation is exactly that; it’s simply considering both the history of past updates and the present force into consideration before deciding the ball trajectory. Pictorially,



In our velocity formulation, $\beta < 1$ is a hyperparameter, similar to α, the learning rate. Two things to note: (1) when $\beta = 0$, we solely rely on the gradient vector to determine our velocity at next step, and (2) the hyperparameter β repeatedly dampens the impact of past on the present, i.e., the most recent past is weighted $\beta\times$, while distant past, say k steps ago, is weighted only by a factor of β^k.

Now, our iteration of gradient descent with momentum can be written as:

$$\theta^{(k+1)} = G(\theta^{(k)}) = \theta^{(k)} - \alpha v^{(k)},$$

where α is the step size, as usual. Now, this may seem like some innocent cheap hack, a simple trick, if you will, to get around gradient descent’s zig-zag behavior. Despite its simplicity, momentum gives a *quadratic* speedup to many functions (Goh, 2017). Now, when universe gives you quadratic speedups, you pay attention.

1.2 Convergence analysis

Let us analyze the (hopefully, improved) convergence of gradient descent with momentum. Again, we will stick with our strictly convex quadratic, but an even simpler one (ignoring vector b and scalar c):

$$J(\theta) = (1/2)\theta^T A\theta,$$

for some symmetric positive definite $A \in \mathbb{R}^{d\times d}$. The gradient can be computed as $\nabla J(\theta^{(k)}) = A\theta$.[3] Now, our iteration is:[4]

$$\theta^{(k+1)} = \theta^{(k)} - \alpha v^{(k)};$$
$$-A\theta^{(k+1)} + v^{(k+1)} = \beta v^{(k)}$$

which can be written more compactly as

$$\begin{bmatrix} 1 & 0 \\ -A & 1 \end{bmatrix} \begin{bmatrix} \theta^{(k+1)} \\ v^{(k+1)} \end{bmatrix} = \begin{bmatrix} 1 & -\alpha \\ 0 & \beta \end{bmatrix} \begin{bmatrix} \theta^{(k)} \\ v^{(k)} \end{bmatrix}.$$

Again, we go back to the notion that the amount of stretch by a matrix is completely determined by the eigenvalues. To this end, let us track an eigenvector of A, say q, with the corresponding eigenvalue λ, i.e., $Aq = \lambda q$. Further, let $\theta^{(k)} = c^{(k)}q$ and $v^{(k)} = d^{(k)}q$ for some scalars $c^{(k)}$ and $d^{(k)}$. This follows that $A\theta^{(k)} = Ac^{(k)}q = c^{(k)}Aq = c^{(k)}\lambda q$; similarly $Av^{(k)} = d^{(k)}\lambda q$. This reduces our matrix form above to track the coefficients as:

$$\begin{bmatrix} 1 & 0 \\ -\lambda & 1 \end{bmatrix} \begin{bmatrix} c^{(k+1)} \\ d^{(k+1)} \end{bmatrix} = \begin{bmatrix} 1 & -\alpha \\ 0 & \beta \end{bmatrix} \begin{bmatrix} c^{(k)} \\ d^{(k)} \end{bmatrix}.$$

Hence, we have

$$\begin{bmatrix} c^{(k+1)} \\ d^{(k+1)} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ \lambda & 1 \end{bmatrix} \begin{bmatrix} 1 & -\alpha \\ 0 & \beta \end{bmatrix} \begin{bmatrix} c^{(k)} \\ d^{(k)} \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & -\alpha \\ \lambda & \beta - \lambda\alpha \end{bmatrix}}_{=R} \begin{bmatrix} c^{(k)} \\ d^{(k)} \end{bmatrix} = R \begin{bmatrix} c^{(k)} \\ d^{(k)} \end{bmatrix}.$$

Observe that after k steps, the starting vector is multiplied by R^k. For fast convergence to zero, which is the minimum of our $J(\theta)$, we need to choose α and β such that R is small. Once again, we recall that the eigenvalues of a matrix determine its stretch, i.e., the eigenvalues of R determine the convergence. Under the specific condition of

$$-1 \le \frac{1+\beta-\lambda\alpha}{2\sqrt{\beta}} \le 1,$$

we find that the magnitude of the eigenvalues of R is $\sqrt{\beta}$.[5] In other words, the rate of convergence of momentum, governed by R, is $\sqrt{\beta}$. To emphasize, this convergence rate applies no matter the step size, *so long as* the above setting of $-1 < (1+\beta-\lambda\alpha)/(2\sqrt{\beta}) < 1$ is satisfied for all eigenvalues λ of A.

This might not seem as magical at a first glance, but as a final step, let us estimate optimal β; since convergence depends on β, we want β to be as small as possible. Since $0 < \lambda_{\min}(A) \le \lambda \le \lambda_{\max}(A)$, the above condition holds iff

$$-1 \le \frac{1+\beta-\lambda_{\max}\alpha}{2\sqrt{\beta}} \quad \text{and} \quad \frac{1+\beta-\lambda_{\min}\alpha}{2\sqrt{\beta}} \le 1.$$

The smallest value of β occurs when inequalities become equalities above, and if you solve for α and β (two equations and two unknowns), you get

$$\alpha = \frac{2+2\beta}{\lambda_{\max}+\lambda_{\min}}; \qquad \sqrt{\beta} = \frac{\sqrt{\lambda_{\max}}-\sqrt{\lambda_{\min}}}{\sqrt{\lambda_{\max}}+\sqrt{\lambda_{\min}}} = \frac{\sqrt{\lambda_{\max}/\lambda_{\min}}-1}{\sqrt{\lambda_{\max}/\lambda_{\min}}+1}.$$

[3]For those less familiar with matrix derivatives, The Matrix Cookbook serves as a good reference: https://www.math.uwaterloo.ca/~hwolkowi/matrixcookbook.pdf.

[4]We write the equation for $v^{(k+1)}$ and not $v^{(k)}$, and rearrange the terms to have $\beta v^{(k)}$ on the right.

[5]See Appendix B for complete proof.

Remember, $\sqrt{\beta}$ is the convergence rate of momentum.

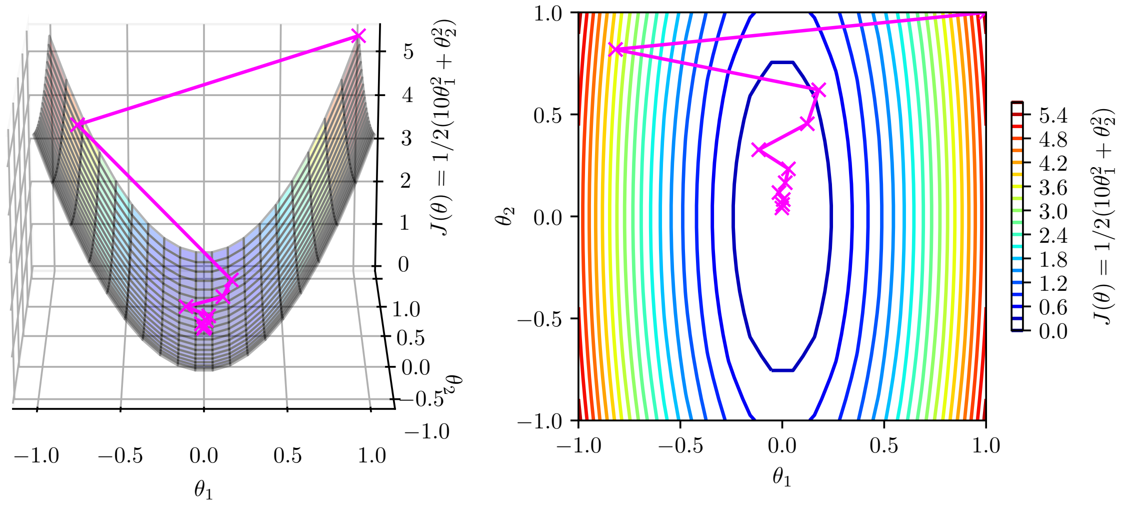
If you don't yet see it, here's the single-sentence takeaway: with momentum, the $\lambda_{\max}/\lambda_{\min}$ factor in the convergence rate of gradient descent is now replaced with $\sqrt{\lambda_{\max}/\lambda_{\min}}$! For the case of $\lambda_{\max}/\lambda_{\min} = 100$, the convergence rate goes from 0.98 without momentum to 0.82 (remember: rates close to 1 mean no progress). Ten steps of ordinary gradient descent multiply the starting error by 0.82, which is matched by a single step with momentum.

1.3 Momentum at play

Now let's run momentum iteration using our previous example with $\lambda_{\max}/\lambda_{\min} = 10$:

$$J(\theta) = \frac{1}{2}(10\theta_1^2 + \theta_2^2).$$

For the same $\alpha = 2/11$ and $\beta = (\sqrt{10} - 1)/(\sqrt{10} + 1)$, we can clearly see below that the momentum helps reduce the "zig-zaggyness" in the updates (although there still exists some—we will come back to this later):



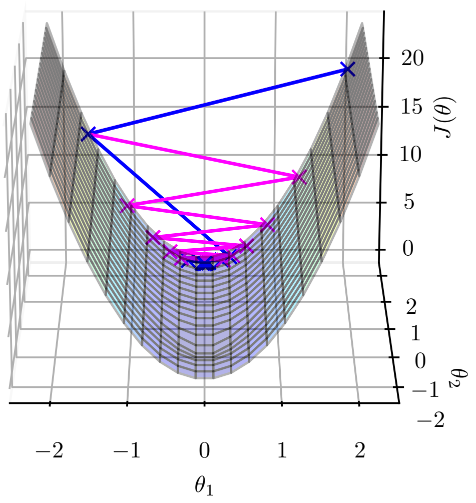
We can clearly see that momentum definitely helps speedup convergence. One final thing to note (and can be easily visualized in the demo[6]) is that unlike gradient descent, which can get stuck in local minima, descent with momentum *can* escape local minima (because the accumulated moment could propel out of a local minimum).

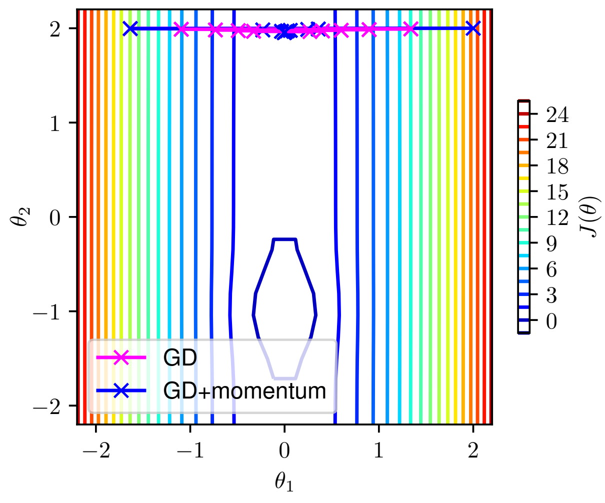
Aside on practical implementations. Of course, the convergence rate and optimal hyperparameters in §1.2 require explicit knowledge of $\lambda_{\min}$ and $\lambda_{\max}$, which is often not known a priori. A common practice is to set β as close to 1 as possible, i.e., rely on the past updates to ensure smooth updates, and then tune α accordingly.

2 Adaptive learning rates

Look at the trajectory of our iterates from the momentum example in §1.3: it looks like our momentum iteration made progress initially but eventually slowed down. Recall that momentum is dampened past velocity, previous updates, that is perturbed by a current external force, gradient. If the gradient is zero in one or more directions, then momentum cannot help us in those directions. To drive this point, let us look at the trajectories of gradient descent and momentum for an extreme example:

[6]Demo materials adapted from: `https://github.com/lilipads/gradient_descent_viz`.





Clearly, both gradient descent and momentum exhibit extremely slow (if any) convergence, once they are only moving along θ_2, which is almost flat. The gradient vector for the function everywhere except at the minima "dip" is

$$\nabla J(\theta^{(k)}) = \begin{bmatrix} 100\theta_1^{(k)} \\ 0.01\theta_2^{(k)} \end{bmatrix}.$$

The gradient vector clearly indicates that our iterative methods favor moving along θ_1 direction more than θ_2 since the gradient along θ_1 is significantly higher than that in θ_2. A simple idea to counter this is: what if take different step sizes along different directions? We will make this more concrete, but for now, let us motivate this problem with a less cherrypicked example.

In previous lectures and projects, we motivated the task of email spam classification using textual features. One way of representing email data as features is to preset a dictionary of words and populating features as indicating the count of a specific word in the dictionary (e.g., the word "machine" appearing in an email 12 times). It is easy to realize that the feature values for commonly-occurring words is going to be high within a training sample and across the training set. As a consequence, gradient descent and momentum tend to optimize for these commonly-occurring words and make little to no progress along the rarer directions.

2.1 AdaGrad

We will ignore momentum for the remainder of this section and show improvements over gradient descent. That said, modern optimizers enable both momentum and adaptive step sizes to facilitate fast convergence; we will say more on this later.

Revisiting what we noted earlier, one idea is to use different step sizes along different directions. Specifically, the more we have updated a certain feature, the less it will be updated in the future, allowing for other features to catch up. In our extreme example, as we proceed through our iteration, we want to update θ_2 more and θ_1 less. Formally, we can track this notion of notion of how much a parameter gets updated using the gradient value:

$$g_i^{(k)} = \underbrace{g_i^{(k-1)}}_{\substack{\text{how much } \theta_i \text{ was updated} \\ \text{in the past}}} + \underbrace{(\nabla J(\theta^{(k)})[i])^2}_{\substack{\text{how much } \theta_i \text{ is being} \\ \text{updated now}}}.$$

In our previous extreme example with with $\nabla J(\theta^{(0)})[1] = 100\theta_1^{(0)}$, making $g_1^{(1)} = (100\theta_1^{(0)})^2$, while $g_2^{(1)} = (0.01\theta_2^{(0)})^2$. Now all we need to do is adapt the step sizes based of $g_i^{(k)}$s, i.e., the

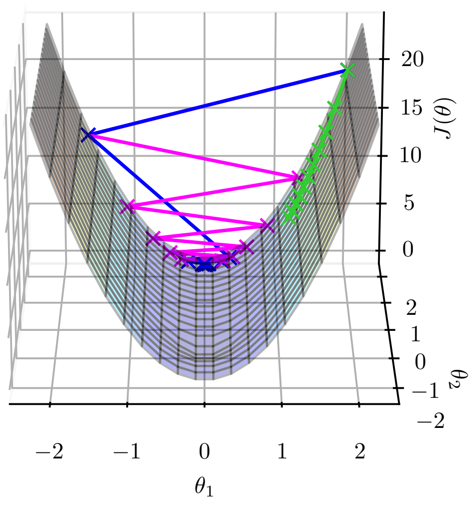
i-th parameter is updated as:

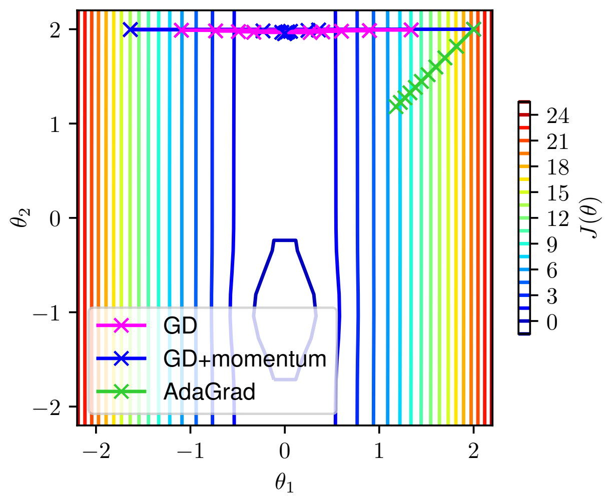
$$\theta_i^{(k+1)} = \theta_i^{(k)} - \frac{\alpha}{\sqrt{g_i^{(k)} + \epsilon}} \nabla J(\theta^{(k)}),$$

where ϵ is some small constant in the order of $\mathcal{O}(10^{-8})$, used to avoid division by zero. One thing worth clarifying here is that when we say "how much θ_i was updated," we mean $\nabla J[i]$ and not $\alpha \nabla J[i]$—this is equivalent to saying that if you've taken a larger step on a flat surface and are still on the flat surface, you take another large step.

2.2 AdaGrad in action

Let's see if AdaGrad helps us with movement along θ_2 direction in our extreme example:

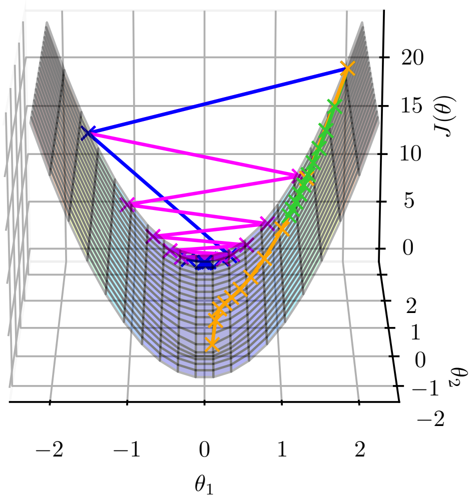


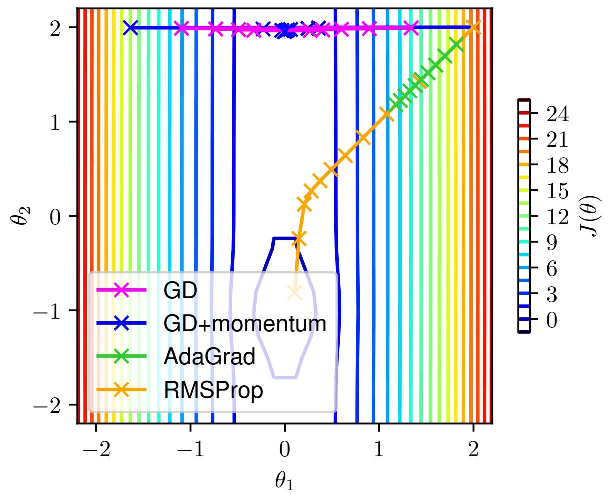


Clearly, we have made progress in the "flatter" direction as well as the other one. That said, it is clear the rate of converge has significantly slowed. A common solution to this—which also allows us to effectively navigate non-convex landscapes where using the *entire* history of both plateaus and valleys might be detrimental—is to use a decay factor:

$$g_i^{(k)} = \rho g_i^{(k-1)} + (1-\rho)(\nabla J(\theta^{(k)})[i])^2,$$

where ρ is some decay factor. This modification is often referred to as the RMSProp (root mean square propagation). With a decay factor of 0.9, we observe significantly faster convergence:





3 Stochastic gradient descent

Let us switch focus to understanding the computational complexity of implementing gradient descent approaches for (large) machine learning models. As motivating thought, let us compute the cost-per-iteration of running gradient descent. For a given training dataset, $\mathcal{D} = \{x^{(j)} | 1 \leq j \leq n\}$, recall that the overall cost function to minimize is of the form

$$J(\theta) = \sum_{j=1}^{n} \ell(x^{(j)}, y^{(j)}; \theta),$$

for some loss function, ℓ. Now, the gradient of $J(\theta)$ can be computed as a simple sum of constituent $\ell(x^{(j)}, y^{(j)}; \theta)$ functions. Simply put,

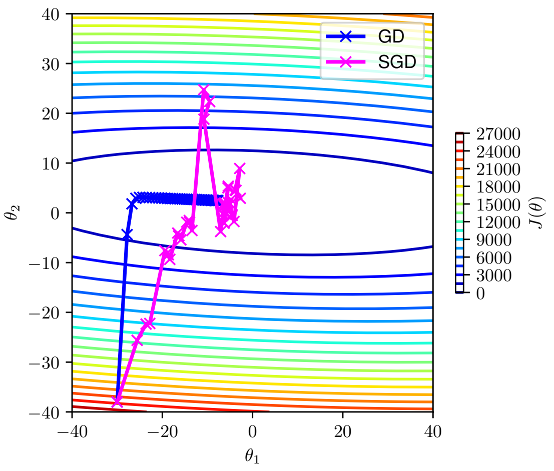
$$\nabla J(\theta) = \sum_{j=1}^{n} \nabla \ell(x^{(j)}, y^{(j)}; \theta).$$

Now for d-dimensional $x^{(j)}$s, we need to compute partials for every dimension, which takes $\mathcal{O}(d)$, and we need to do this for all data points, bringing the overall complexity of computing the gradient to $\mathcal{O}(nd)$. If we run gradient descent for a total of T iterations, the overall computational complexity becomes $\mathcal{O}(ndT)$. We observe that gradient descent needs to do $\mathcal{O}(nd)$ work before making a single update to the parameters. This is especially problematic because the larger the dataset, the slower the optimization is! Can we do better?

A rather basic idea is to randomly sample a *mini-batch* of training samples at a given optimization step, and use that to approximate $J(\theta)$ in that step; choose a different mini-batch in the next step, and continue till convergence. In the simplest setting, we can set mini-batch size to be one, i.e., we randomly select one training sample, say $\tilde{j}$, and use $\ell(x^{(\tilde{j})}, y^{(\tilde{j})}; \theta)$ as our noisy estimator for $J(\theta)$. Our iteration can be written as:

$$\theta^{(k+1)} = \theta^{(k)} - \alpha \nabla_{\tilde{j}} J(\theta^{(k)})$$
$$\nabla_{\tilde{j}} J(\theta^{(k)}) = \ell(x^{(\tilde{j})}, y^{(\tilde{j})}; \theta^{(k)}).$$

(For obvious reasons,) we call the above "stochastic" gradient descent. Note that the cost-per-iteration is only $\mathcal{O}(d)$, though the number of iterations might be larger than that for gradient descent. An example comparing gradient descent and stochastic gradient descent trajectories is shown to the right.



While the convergence proof is out of the scope for this class, we want to remark a few properties related to convergence of stochastic gradient descent. A typical feature of using such a noisy estimator is that it can jump out of a local minima, which is easy to realize. A more intriguing property is that stochastic gradient descent offers what is often known as a "semi-convergence": fast convergence to the optimal parameters at start (faster than gradient descent), while the later iterations, especially closer to the optima are more erratic. A common strategy here is to have adaptive step sizes, which would affect the convergence rate, or resort to early stopping.

As a final note, all the variants discussed above for gradient descent can be applied to stochastic gradient descent as well.

4 Conclusion

To recap, in this lecture, we saw several improvements over vanilla gradient descent. First, we noted that vanilla gradient descent can converge rather poorly when $\lambda_{\max}/\lambda_{\min}$ is really large, implying narrow, steep bowls. To counter this, we saw heavy-ball momentum, where we gained momentum when the past and current gradients aligned and slowed down otherwise. Consequently, we showed how this seemingly innocent change allowed us to go from running one momentum step for every ten steps of gradient descent (when $\lambda_{\max}/\lambda_{\min} = 100$). Next, we realized how movement along flatter surfaces is quite ineffective when just using gradients. Here, we devised a simple scheme of adaptive learning rates by decreasing step sizes proportional to the curvature of the objective function, AdaGrad, and its faster variant, RMSProp.

Finally, we looked at computational considerations of implementing gradient descent, and presented a noisier estimator which allowed for lower cost-per-iteration.

Adam optimizer. It would be remiss not to mention Adam (Kingma and Ba, 2014, adaptive moment estimation; with over $206,930$ citations and counting), the optimizer that dominates large language model training. Adam combines the best of both worlds: momentum and RMSProp. While we do not cover Adam in these lecture notes, its underlying principles are well-studied here, and we leave it as a topic for further reading.

A Notation

$\mathcal{D}$	The training dataset of n samples
n	The number of training samples in the dataset $\mathcal{D}$
d	The number of parameters we wish to estimate. Note that we also use d to represent the number of features; these need not be the same, as we will see later in the course
$x^{(j)} \in \mathbb{R}^d$	The d-dimensional feature vector associated with the j-th training sample
$y^{(j)}$	The class label associated with the j-th training sample
$x_\ell^{(j)} \in \mathbb{R}$	The ℓ-th element of $x^{(j)}$
$\theta \in \mathbb{R}^d$	A vector of d model parameters
$\theta_\ell \in \mathbb{R}$	The ℓ-th element of θ
$J(\theta)$	The cost function we are trying to optimize (here, minimize)
$\ell(x^{(j)}, y^{(j)}; \theta)$	The loss evaluated by running a model parameterized by θ on $x^{(j)}$ and comparing it to the true label, $y^{(j)}$
$\theta^{(k)} \in \mathbb{R}^d$	The k-th iterate of model parameters
$\theta^\star \in \mathbb{R}^d$	The optimal model parameters that minimize $J(\theta)$
G	The iteration that updates the parameters based on their current values. At the optima, $\theta^\star$, we require $G(\theta^\star) = \theta^\star$
$\nabla J(\theta^{(k)}) \in \mathbb{R}^d$	A vector of partial derivatives of $J(\theta)$, evaluated at $\theta^{(k)}$, known as the gradient vector (read: "nabla J" or "del J"). Advanced: If you're familiar with Jacobian, gradient is the dual of Jacobian for scalar-valued functions.
α	The learning rate or step size used to move along the steepest descent direction in gradient descent
$\varepsilon^{(k)}$	Error at step k, computed as $\theta^\star - \theta^{(k)}$
$\lVert u \rVert$	The (two) norm of a vector u
λ_j	The j-th eigenvalue of a matrix
$\lambda_{\min}$	The smallest eigenvalue of a matrix
$\lambda_{\max}$	The largest eigenvalue of a matrix

B Computing eigenvalues of iteration matrix

Let ν be an eigenvalue of R; we can obtain the eigenvalues by solving the characteristic polynomial, $\det(\nu \mathrm{I} - R) = 0$. Hence, we have

$$\begin{aligned}\det(\nu \mathrm{I} - R) = \det\left(\nu \mathrm{I} - \begin{bmatrix} 1 & -\alpha \\ \lambda & \beta - \lambda\alpha \end{bmatrix}\right) &= \det\left(\begin{bmatrix} \nu - 1 & -\alpha \\ \lambda & \nu - \beta + \lambda\alpha \end{bmatrix}\right) \\ &= (\nu - 1)(\nu - \beta + \lambda\alpha) + \lambda\alpha \\ &= \nu^2 - \beta\nu + \lambda\alpha\nu - \nu + \beta - \lambda\alpha + \lambda\alpha \\ &= \nu^2 - (1 + \beta - \lambda\alpha)\nu + \beta \overset{\text{set}}{=} 0.\end{aligned}$$

So, from the quadratic formula, the eigenvalues of R are

$$\nu = \frac{(1 + \beta - \lambda\alpha) \pm \sqrt{(1 + \beta - \lambda\alpha)^2 - 4\beta}}{2}.$$

Next, we will operate under a specific condition of the discriminant (the term within the square root) is negative. That is,

$$(1+\beta-\lambda\alpha)^2 - 4\beta < 0, \text{ or } -1 < \frac{1+\beta-\lambda\alpha}{2\sqrt{\beta}} < 1.$$

When the discriminant is negative, the roots of the quadratic are complex conjugates, say, ν and $\bar{\nu}$, with $|\nu| = |\bar{\nu}|$.

One final thing we recall is that the determinant of a matrix is the product of its eigenvalues. For R, we have

$$\det(R) = \beta - \lambda\alpha + \lambda\alpha = \beta.$$

Since $\det(R)$ is the product of eigenvalues of R, ν and $\bar{\nu}$, we have $\beta = \nu\bar{\nu} = |\nu|^2$, which gives us $|\nu| = \sqrt{\beta}$. Hence, under the specific condition of

$$-1 < \frac{1+\beta-\lambda\alpha}{2\sqrt{\beta}} < 1,$$

we have that the magnitude of eigenvalues of R to be $\sqrt{\beta}$.

References

G. Goh. Why momentum really works. *Distill*, 2017. doi: 10.23915/distill.00006. URL http://distill.pub/2017/momentum.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014. URL https://arxiv.org/abs/1412.6980.

G. Strang et al. *Linear algebra and learning from data*, volume 4. Wellesley-Cambridge Press Cambridge, 2019. Read Part VI.4: "Gradient Descent Toward the Minimum" (pp. 348–353).

R. Zadeh. Lecture 11: Distributed Algorithms and Optimization. *CME 323 lecture notes, Spring 2015*, 1(1):1–5, 2015. URL https://stanford.edu/~rezab/classes/cme323/S15/notes/lec11.pdf. Scribed by K. Bergen, K. Chavez, A. Ioannidis, and S. Schmit.

(Last compiled: 2/27/2025, 12.51 Noon ET.)