Lecture 2: Continuous Optimization in Deep Learning

Lecturer: Aleksander Mądry *Scribes: Scott Foster, Luke Kulik, Kevin Li*
(Revised by Andrew Ilyas and Dimitris Tsipras)

1 Introduction

In the previous lecture, we covered *gradient descent* methods. An iterative approach to solving unconstrained minimization problem whose update step Δ^t, derived from a locally linearization of the (β-smooth) function f ended up being:

$$\Delta^t := \arg\min_{\Delta} \nabla f\left(x^t\right)^\top \Delta + \frac{1}{2}\beta\|\Delta\|^2 \qquad = -\frac{1}{\beta}\nabla f\left(x^t\right),$$

and our next iterate became:

$$x^{t+1} := x^t + \Delta^t.$$

We have also shown that, once the function f is also α-strongly convex, for some $\alpha > 0$ then, we have that, for any $\varepsilon > 0$,

$$f(x^T) - f(x^*) \leq \varepsilon,$$

once

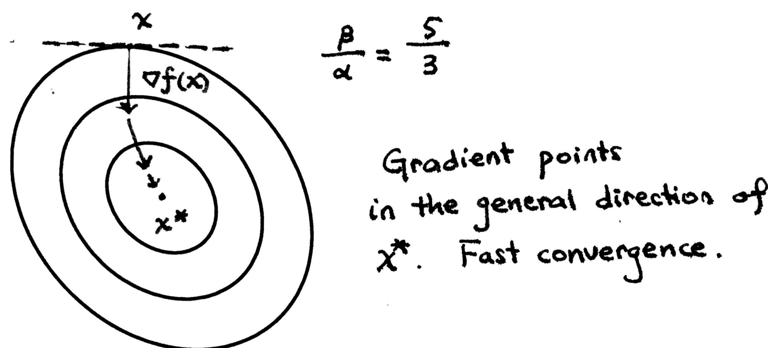
$$T = C \cdot \left(\frac{\beta}{\alpha}\log\frac{R}{\varepsilon}\right) = C \cdot \left(\kappa \log\frac{R}{\varepsilon}\right), \tag{1}$$

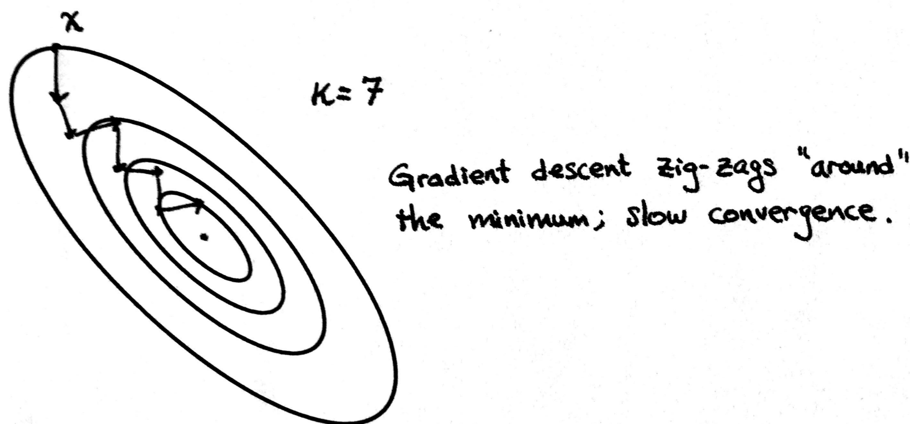
for some constant $C > 0$. Here, the quantity $\kappa := \beta/\alpha$ is called the condition number of f. It can be viewed as a reflection of the "badness" of the (Euclidean) geometry of f.

In particular, if f is twice differentiable everywhere (which we assumed here), the values of α and β correspond to the bounds on the smallest and largest eigenvalues of the Hessian. That is,

$$\alpha = \inf_x \lambda_{\min}(\nabla^2 f(x)) \quad \text{and} \quad \beta = \sup_x \lambda_{\max}(\nabla^2 f(x)) \tag{2}$$

Now, to get some intuition regarding why this condition number is important one should note that when $\kappa = 1$ (i.e., when f is quadratic), the gradient point directly in the direction of the minimizer. Conversely, when κ is large, the gradient direction does not correlate well with the direction towards minimum. As a result, the optimization path tends to slowly zig-zag toward the minimizer. See the figures below.





1.1 Momentum Gradient Descent

So, we see that the dependence of the convergence of our algorithm on the condition number κ is, in some sense, inevitable. But does this dependence have to be linear?

One can show that, in the worst-case, it will be linear for the gradient descent algorithm we considered. But maybe there is a different variant of this algorithm that does better?

Indeed! There is. The family of algorithms to consider here is called momentum gradient descent methods. These algorithms attempt to overcome the zig-zagging problem we alluded to above by working with the following dynamics:

$$\begin{aligned} v^{t+1} &= \gamma v^t - \eta \nabla f(x^t) \\ x^{t+1} &= x^t + v^{t+1} \end{aligned} \tag{3}$$

One can view this dynamics as corresponding to a physical system of heavy ball with friction. Specifically, here, v is the velocity of the ball, $\gamma < 1$ is the friction parameter and the gradient corresponds to the applied force. The intuition why this kind of dynamics might be helpful is that zig-zagging along the directions orthogonal to the direction of the minimum point will cancel out, and the velocity will instead build up in the desired direction of the minimum.

Indeed, one can show that a certain "predictive" variant of this dynamics, known as Nesterov's momentum variant of gradient descent, given by

$$\begin{aligned} v^{t+1} &= \gamma_t v^t - \eta_t \nabla f(x^t + \gamma_t v^t) \\ x^{t+1} &= x^t + v^{t+1}, \end{aligned} \tag{4}$$

with a very specific way of setting γ_t and η_t, can be formally analyzed and shown to achieve ε-optimality in only

$$T = O\left(\sqrt{\kappa} \log \frac{R}{\varepsilon}\right) \text{ steps.} \tag{5}$$

Also, one can show that if the interaction with f is restricted to its gradients only (so-called first order optimization model), then the square root dependence on κ is the best asymptotically possible bound (in the worst case).

2 Generalized gradient descent

In machine learning applications, we typically are able to learn more about f than its gradient—can we use this extra information to achieve better bounds? Recall that the above analysis relied on the sandwiching inequality

$$\frac{1}{2}\alpha\|\Delta\|^2 \leq \varsigma_x(\Delta) \leq \frac{1}{2}\beta\|\Delta\|^2, \quad \text{for all } x \text{ and } \Delta. \tag{6}$$

The norm used in the above equation so far was, naturally, assumed to be an Euclidean norm. However, it does not have to be! We can change the norm $\|\cdot\|$ so that, in effect, the constants α and β are improved. And what we will get in this way is so-called *general gradient descent method*.

There is many norm choices we might consider here. But the one family of norms that will be most useful to us will be so-called *A-norm* $\|\cdot\|_A$ specified by some positive definite matrix $A \succ 0$ and defined as

$$\|v\|_A := v^\intercal A v.$$

Note that if A is an identity matrix, this corresponds directly to the Euclidean norm.

Now, in analogy to the "standard" gradient descent setup, we minimize the objective

$$\underset{\Delta}{\text{minimize}}\ \varphi_x(\Delta) + \beta\|\Delta\|_A^2. \tag{7}$$

Note the solution to (7) is given by

$$\frac{1}{\beta}A^{-1}\nabla f(x) = \arg\min_{\Delta} \varphi_x(\Delta) + \frac{1}{2}\beta\|\Delta\|_A^2. \tag{8}$$

So, it is *not* necessarily corresponding to moving in the direction of the actual gradient.

The right choice of A will be helpful as it can "rescale" the ellipsoidal level sets of f (i.e., which correspond to large κ) to make them much "rounder" and thus enable more rapid convergence. (Although at the cost of having to compute the inverse of A, or solving a linear system in it each time we take a step.)

Now, one could wonder what is the "best" choice of A. This choice, in a sense, turns out to be taking A to be the Hessian $\nabla^2 f(x)$ of our function at the point x. (Note that it means this might be a different norm at different points!) This choice is motivated by noticing that if Δ is "not too big" we have that

$$\varrho_x(\Delta) \lessapprox \frac{1}{2}\Delta^\intercal \nabla^2 f(x)\Delta \approx \frac{1}{2}\|\Delta\|_{\nabla^2 f(x)}^2. \tag{9}$$

So, the (local) condition number here is close to 1 and thus being best possible.

The resulting algorithm is known as *Newton's method* and its update rule ends up being

$$x^{t+1} = x^t - \eta\nabla^2 f(x)^{-1}\Delta f(x). \tag{10}$$

Here, the step size η have to be chosen carefully to address the requirement that Δ needs to be "not too big". In general, analyzing the convergence of such methods turns out to be tricky (also, because of the norm potentially changing in each iteration) and is beyond the scope of this class. But it can lead to significant speed ups. Unfortunately, this comes at a price of the need to invert the Hessian (or rather solve linear system in it), which tends to be computationally expensive (and also might require too much space to even store the Hessian explicitly). This is particularly relevant in deep learning context.

2.1 Quasi-Newton methods

The idea behind so-called quasi-Newton methods is to try to strike a compromise between the benefit of being able to improve the geometry of the problem via rescaling and the computation time/space constraints. The basic idea is to use the update rule

$$x^{t+1} = x^t - \eta H_t^{-1}\nabla f(x) \tag{11}$$

for a sequence of matrices $H_t \succ 0$ that approximates the Hessian (c.f. (10)) in some way.

We discuss two choices of approximations, the **BFGS** (as well as its variant **L-BFGS**) and the AdaGrad algorithm.

2.2 BFGS method

In the BFGS method we aim to approximate in certain sense the *inverse* of the Hessian. Specifically, we start with a "trivial" guess for the inverse of the Hessian: the identity matrix, i.e., $B_0 = I$. Then, in each step, we refine this guess by updating it based on the new "information" we got about the Hessian. (Note that, importantly, we make an implicit assumption here that Hessians at different point are the same/close to each other.)

Note that the inverse Hessian and thus our guess for it B_t at time t, has to satisfy a number of simple conditions. First of all, it has to be positive definite, i.e., $B_t \succ 0$. Secondly, if we observe how the gradients change between two consecutive points, it has to "explain" this change. That is, we need to have that

$$B_t y_t = s_t \tag{12}$$

where $s_t = x^t - x^{t-1}$ and $y_t = \nabla f(x^t) - \nabla f(x^{t-1})$.

This constraints restrict the choice of B_t but do not uniquely define it. To this end, we settle on a choice of B_t that satisfies these constraints and is "minimal" in the sense of its similarity to our previous estimate B_{t-1}. Specifically, we take it to be

$$B_t = \min_B \|B - B_{t-1}\|_F \text{ subject to } B_t y_t = s_t \text{ and } B_t \succ 0 \tag{13}$$

where $\|\cdot\|_F$ is the Frobenius norm.

Once again, the implicit assumption in this algorithm is that the Hessian is globally constant; indeed, it is possible to show that BFGS accelerates the optimization of quadratic functions. This is clearly not true in applications, yet this method works well in practice (sometimes).

In high dimensional problems, even storing the Hessian is not really feasible though. So, a simplified version of the BFGS method, known as L-BFGS ("limited memory" BFGS) is a variant where (13) is replaced with

$$B_{t+1} = \min_B \|B - I\|_F \quad \text{ subject to } B_t y_t = s_t \text{ and } B_t \succ 0, \tag{14}$$

i.e., we assume always that B_{t-1} is just an identity matrix. The L-BFGS heuristic is also successfully used in practice, even though its theoretical motivation seems to be much weaker.

3 AdaGrad

The final algorithm we consider is the AdaGrad algorithm, which is derived using the so-called online convex optimization (or online learning) framework. In this framework, one considers the following iterative online prediction game:

- For each $t = 1, \ldots, T$:
  1. output a choice x_t;
  2. learn a "penalty" function f_t and incur penalty $f_t(x_t)$ corresponding to your choice.

The goal of this game is to choose a sequence of choices so that to minimize the sum of all the corresponding penalties $\sum_{t=1}^T f_t(x_t)$.

Of course, since the penalty function f_t is revealed only after the choice x_t was made, the resulting penalty can be arbitrary large. So, there is no hope one would be able to provide any non-trivial guarantees for a given strategy of making the choices in absolute terms. Still, it turns out that there is a useful measure of *relative* performance: the *regret*, which is defined as

$$R(T) := \sum_t f_t(x_t) - \min_x \sum_t f_t(x). \tag{15}$$

That is, R is the difference between the penalty of our algorithm against a hypothetical algorithm which knows the sequence $\{f_t\}$ in advance but is restricted to choosing a single choice x^* in each one of the rounds.

Note that minimizing $R(T)$ with $f_1 = \cdots = f_T$, $f := f_T$, amounts to finding a minimum of f. In particular, if $R(T)$ is sub-linear, i.e., $R(T) \in o(T)$, then x_t is making progress towards the minimum:

$$R(T) \in o(T) \implies \frac{1}{T}\sum_t f(x_t) - f(x^*) \to 0, \quad \text{where } f(x^*) = \min f(x). \tag{16}$$

Follow the regularized leader. One algorithm that achieves sublinear regret is the *regularized follow the leader* algorithm, defined by always playing

$$x^t := \arg\min_x \left[\eta \sum_{s=1}^{t-1} f_s(x) + \|x\|^2\right], \tag{17}$$

and taking $x_0 = 0$.

For appropriate choices of the parameter η, we can show that

$$R(T) \le O\left(\|x^* - x^0\| \sqrt{\sum \|\nabla f_t(x_t)\|}\right) \in o(T), \tag{18}$$

as required.

The AdaGrad algorithm. Now, in analogy to what we did above, we could make this algorithm be tuned to the geometry of the problem by introducing a different (to Euclidean) norm $\|\cdot\|_{H_t}$ in each step. Specifically, we could have

$$x^t := \arg\min_x \left[\eta \sum_{s=1}^{t-1} f_s(x) + \|x\|^2_{H_t}\right] \quad \text{in which case} \quad R(T) \le O\left(\|x^* - x^0\| \sqrt{\sum \|\nabla f_t(x_t)\|_{H_t}}\right). \tag{19}$$

It turns out that we can achieve the tighter bound

$$R(T) \le \min_{H \succ 0} O\left(\|x^* - x^0\| \sqrt{\sum \|\nabla f_t(x_t)\|_H}\right), \quad \text{the minimum over positive definite } H \tag{20}$$

if we set H_t to the square-root of the empirical covariance matrix

$$H_t = \left(\sum_{s=1}^{t-1} \nabla f_s(x_s) \nabla f_s(x_s)^T\right)^{1/2}. \tag{21}$$

This latter choice is the AdaGrad algorithm. In machine learning, the covariance matrix may be too large to fit inside memory, so another choice is to use the only the diagonal entries

$$\tilde{H}_t = \left(\sum_{s=1}^{t-1} \operatorname{diag} H_t\right)^{1/2}. \tag{22}$$

One benefit of (21) (and also (22)) is that it allows different coordinates of x to have different learning rates. Specifically, the learning rate for each parameter depends upon how large the magnitude of gradients along that parameter dimension has been in the previous iterations. For each parameter dimension, AdaGrad accumulates (as a vector r the square of magnitudes of gradient g along that dimension in all the previous iterations ($r_{i+1} \leftarrow r_i + g_i^2$) and divides the learning rate by square root of r_i along each dimension. The resulting update step can then be written as:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \operatorname{diag}\left(\sum_{\tau=1}^{t-1} g_\tau g_\tau^\top\right)^{-1/2} g_t \qquad \text{where } g_\tau = \nabla^{(\tau)} f(\theta_\tau), \text{ gradient collected at time } \tau \tag{23}$$

As such, AdaGrad heavily reduces the learning rate for parameters that have too large oscillations (as indicated by high-magnitude gradients), and has much less of an effect on the learning rate for parameters that do not contribute much to the oscillations (i.e. whose gradient magnitude decreases).

This feature is especially useful in cases coordinates of the data set have different rates of occurrence. Consider a natural language setting where

$$x = (\text{binary vector of 0's and 1's})_{v \in V} \tag{24}$$

with one word v for every word in the vocabulary V. If v is a common word, then there will be many 1's in the vth column (e.g., '*and*') which may not be very informative; on the other hand, a less common word w may be very informative (e.g., '*solipsism*'). A different learning rate for every coordinate allows us to "slow down" the not very helpful learning on the vth coordinate without hampering the helpful learning on the wth one.

3.1 RMSProp Algorithms

In the above, r_i is an accumulator of non-negative terms (magnitudes), and hence always increases. As a result, the learning rates for all the parameters decrease, and may in fact vanish (become infinitesimally small). If the learning rate vanishes too quickly, AdaGrad will not be able to learn a value for the parameter and it will remain fixed and far from the optimum. (Note that such a concern only arises because of the non-convex nature of training deep neural networks—in the convex case, AdaGrad is proven to converge.) The RMSProp algorithm[1] was designed as an attempt to circumvent this issue.

Rather than using an accumulator of the squared gradients, RMSProp uses *weighted* sum of square magnitudes of gradients, where the weights exponentially decay (with rate ρ, a hyperparameter) over time. Thus, gradients of more recent iterations are assigned more weight than those in earlier iterations.

Intuitively, one can think of RMSProp as directly addressing the non-convexity of the training landscape. In particular, one might imagine that the steepness/shallowness of this landscape may change along the trajectory from initialization to final weights. By assigning more weight to recent gradient magnitudes, one can view RMSProp as implementing AdaGrad "locally," i.e. according to the current shape of the landscape. This view may help to explain why RMSProp tends to outperform AdaGrad in practice. Concretely, the update step implemented by RMSProp is as follows:

$$\theta^{(t+1)} = \theta^{(t)} - \eta \operatorname{diag}\left(\sum_{\tau=1}^{t-1} \rho^{t-\tau} g_\tau g_\tau^\top\right)^{-1/2} g_t,$$

with g_τ being defined as in (23) and $\rho < 1$ being a user-defined hyperparameter.

3.2 Adaptive Moment Estimate Adam

The Adam [KB14] method, short for "ADAptive Moment" estimation, is a combination of ideas from RMSProp and Nesterov's momentum. The motivation is similar to that of AdaGrad and RMSProp , i.e. reducing the learning rate along dimensions contributing to high oscillation by taking into account historical gradient magnitude. Adam combines this idea with momentum by exponentially-weighted averaging both the first and second "moments" (the gradient g_t and its outer product $g_t g_t^\top$, respectively). The algorithm also introduces "bias correction terms" $1-\beta_1^t$ and $1-\beta_2^t$, which are meant to counteract the fact that we initialize the average squared magnitude as $\boldsymbol{r} = \mathbf{0}$. It turns out that this simple unification of RMSProp /AdaGrad is highly effective in many settings—since its inception Adam has attained widespread popularity (the original paper has 6000+ citations).

Despite this, Adam is still somewhat poorly understood from an optimization perspective. The original paper provides "recommended hyperparameters" that are not theoretically justified but seem to work well in practice on a wide variety of tasks. Furthermore, though the original paper presented a proof of convergence for convex functions, the proof was later shown to be incorrect, and a counterexample of convergence was recently published [RKK18] for a simple convex problem.

In fact, we still lack conclusive evidence that Adam provides significant benefit over stochastic gradient descent in a general sense (although it has been empirically shown to help in many specific cases).

[1] Interestingly, the RMSProp algorithm was never published—it was originally described by Geoff Hinton in a Coursera lecture.

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
Initialize 1st and 2nd moment variables $\boldsymbol{s} = \boldsymbol{0}$, $\boldsymbol{r} = \boldsymbol{0}$
Initialize time step $t = 0$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
$t \leftarrow t + 1$
Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1) \boldsymbol{g}$
Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2) \boldsymbol{g} \odot \boldsymbol{g}$
Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
Compute update: $\Delta \boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
end while

Figure 1: The Adaptive Moment (Adam) algorithm for first-order optimization.

References

[KB14] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. *CoRR*, abs/1412.6980, 2014.

[RKK18] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the Convergence of Adam and Beyond. In *International Conference on Learning Representations*, 2018.