Lecture 8: Accelerating SGD with preconditioning and adaptive learning rates.

CS4787 — Principles of Large-Scale Machine Learning Systems

Recall: Optimization problems can be poorly conditioned when the condition number κ is large. We said last time that intuitively, we'd like to *set the step size larger for directions with less curvature, and smaller for directions with more curvature*. But we couldn't do this with plain GD or SGD, because there is only one step-size parameter. Last time, we talked about **momentum** which addresses this problem by using a momentum term that amplifies the gradient in directions that are consistently the same sign and dampens the gradient in directions that are reversing sign. Today we'll talk about two other methods for addressing the issue of conditioning: preconditioning, and adaptive learning rates.

Preconditioning.

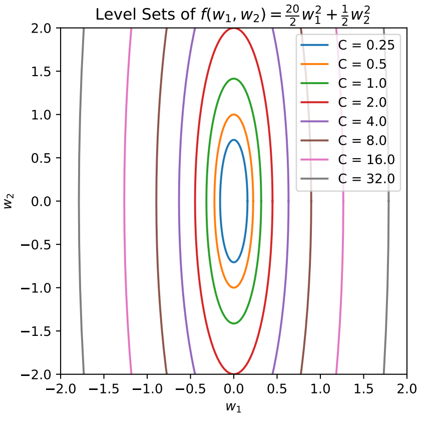
Motivation: one way to think of a large condition number is in terms of how it affects the *level sets* of the optimization problem. For example, if we look at our poorly conditioned optimization problem example from last time, the two-dimensional quadratic

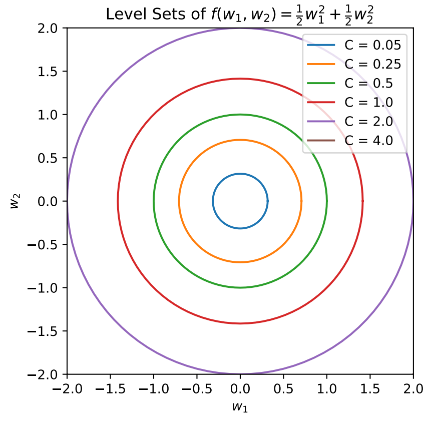
$$f(w) = f(w_1, w_2) = \frac{L}{2} w_1^2 + \frac{\mu}{2} w_2^2 = \frac{1}{2} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}^T \begin{bmatrix} L & 0 \\ 0 & \mu \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}.$$

Here, the level sets of f are the sets on which $f(w) = C$ for some constant C, which take the form

$$2C = L w_1^2 + \mu w_2^2;$$

these are just ellipses. When the problem is poorly conditioned, these ellipses are highly distorted (very far away from being circles). On the other hand, when $\kappa = 1$, the level sets are exactly circles.





Main idea of preconditioning: rescale the underlying space we're optimizing over to make the level sets more like circles. To do this for an objective function f, let's imagine solving the modified optimization problem where we minimize $g(u) = f(Ru)$ for some fixed matrix R. Gradient descent for this task looks like

$$u_{t+1} = u_t - \alpha \nabla g(u_t) = u_t - \alpha R^T \nabla f(R u_t).$$

If we multiply both sides by R, and let $w_t = Ru_t$, we get

$$w_{t+1} = Ru_{t+1} = Ru_t - \alpha RR^T \nabla f(Ru_t) = w_t - \alpha RR^T \nabla f(w_t).$$

Effectively, we're just running gradient descent with gradients scaled by some positive semidefinite matrix $P = RR^T$ (why is it positive semidefinite?). This method is called *preconditioned gradient descent*, and we can apply the same idea to precondition SGD.

Activity: What would be the best preconditioner P to choose for our two-dimensional quadratic example of f above? Is the best preconditioner unique?

How to make preconditioning efficient. In general, a preconditioning matrix P over models of dimension d will take d^2 memory to store, and it will take $O(d^2)$ time to compute the matrix-vector product needed to run preconditioned gradient descent. This can add up to be very expensive, especially when the model size is large. One common way to address this is to use a *diagonal preconditioner*: we restrict P to be a diagonal matrix.

How much memory is needed now to store P? How much time is needed to multiply by P in the preconditioned GD update step?

How to choose the preconditioner. This is all very interesting, but it requires us to pull a matrix P from somewhere. How do we pick a useful P?

- One way: use your intuition about the problem. You may know from the formula for the loss that some dimensions are more curved than others (e.g. the two-dimensional quadratic example above). **Downside: doesn't scale.**
- Another way: use statistics from the dataset. For example, for a linear model you could precondition based on the variance of the features in your dataset.
- Another way: use information from the matrix of second-partial derivatives. For example, you could use a preconditioning matrix that is a diagonal approximation of the Newton's method update at some point. These methods are sometimes called Newton Sketch methods.

Adaptive Learning Rates. Idea: adjust the learning rate per-component dynamically at each timestep based on observed statistics of the gradients. Explicitly,

$$w_{j,t+1} = w_{j,t} - \alpha_{j,t} \left(\nabla f_{\tilde{i}_t}(w_t) \right)_j$$

where $w_{j,t}$ denotes the ith entry of the model at time t, and the learning rate $\alpha_{j,t}$ is now (1) allowed to be different per-parameter, and (2) allowed to vary as a function of the previously observed gradient samples. There are many different schemes for adaptive learning rates.

AdaGrad. AdaGrad sets the step size for each parameter to be inversely proportional to the square root of the sum of all observed squared values of that component of the gradient.

Algorithm 1 AdaGrad

```
input: learning rate factor $\alpha$, initial parameters $w$.
initialize $r \leftarrow 0$
loop
    sample a stochastic gradient $g \leftarrow \nabla f_{\tilde{i}_t}(w)$
    accumulate second moment estimate $r_j \leftarrow r_j + g_j^2$ for all $i \in \{1, \ldots, d\}$
    update model $w_j \to w_j - \frac{\alpha}{\sqrt{r_j}} \cdot g_j$
end loop
```

(Typically we also need to add some small correction factor to avoid dividing by zero in this expression if r_j is zero.) The motivation behind AdaGrad: think about the "optimal" step size rule we derived for convex SGD earlier, where we added a constant amount to the inverse of the step size at each step. (This gave us a $1/t$ step size scheme.) Here, we are also adding a roughly-constant amount to the inverse of the step size at each step, except it's proportional to the magnitude of the gradient sample in that direction. This causes our step sizes to be larger in directions in which the gradient tends to be smaller and vice versa. One problem with AdaGrad: it does not necessarily work well in the non-convex setting, because the learning rate is dependent on the whole history of the algorithm, and for non-convex optimization the trajectory may have passed through regions of very different curvature. This could lead to a step size that is very small in some directions in which we don't want the step size to be small.

RMSProp. A modification to AdaGrad that uses an exponential moving average instead of a sum. This can be more effective for non-convex optimization problems such as neural networks. One potential downside: the step size generally does not go to zero with RMSProp, so we could converge to a noise ball.

Algorithm 2 RMSProp

```
input: learning rate factor $\alpha$, decay rate $\rho$, initial parameters $w$.
initialize $r \leftarrow 0$
loop
    sample a stochastic gradient $g \leftarrow \nabla f_{\tilde{i}_t}(w)$
    accumulate second moment estimate $r_j \leftarrow \rho r_j + (1 - \rho) g_j^2$ for all $i \in \{1, \ldots, d\}$
    update model $w_j \to w_j - \frac{\alpha}{\sqrt{r_j}} \cdot g_j$
end loop
```

Adam. Modified RMSProp to (1) use momentum with exponential weighting, and (2) correct for bias to estimate the first-order and second-order moments of the gradients.

Algorithm 3 Adam

```
input: learning rate factor $\alpha$, decay rates $\rho_1$ and $\rho_2$, initial parameters $w$.
initialize $r \leftarrow 0$
initialize timestep $t \leftarrow 0$
loop
    $t \leftarrow t + 1$
    sample a stochastic gradient $g \leftarrow \nabla f_{\tilde{i}_t}(w)$
    accumulate first moment estimate $s_j \leftarrow \rho_1 s_j + (1 - \rho_1) g_j$ for all $i \in \{1, \ldots, d\}$
    accumulate second moment estimate $r_j \leftarrow \rho_2 r_j + (1 - \rho_2) g_j^2$ for all $i \in \{1, \ldots, d\}$
    correct first moment bias $\hat{s} \leftarrow \frac{s}{1-\rho_1^t}$
    correct second moment bias $\hat{r} \leftarrow \frac{r}{1-\rho_2^t}$
    update model $w_j \rightarrow w_j - \frac{\alpha}{\sqrt{\hat{r}_j}} \cdot \hat{s}_j$
end loop
```

Why does Adam's bias correction work?

How can we think of these methods as relating to diagonal preconditioning?