Recall: Optimization problems can be poorly conditioned when the condition number κ is large.

Last time → momentum
Today → preconditioning, adaptive learning rates

Preconditioning

Motivation: How does κ impact the level sets of the optimization problem.

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

$$\nabla f(w) = \begin{bmatrix} L w_1 \\ w_2 \mu_2 \end{bmatrix} \quad , \quad \nabla^2 f(w) = \begin{bmatrix} L & 0 \\ 0 & \mu \end{bmatrix}$$

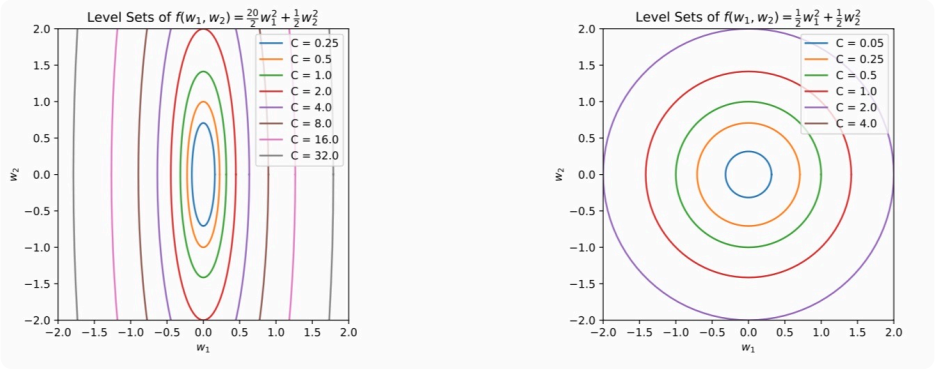
"f L-Lipschitz, μ strongly convex" $\quad L > \mu > 0$

$\kappa = L/\mu$

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

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

These are ellipses.
When poorly conditioned, ellipses highly distorted.
When $\kappa = 1$, 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.

Aside

(1) $\min_{x \in \mathbb{R}^d} f(x) = \min_{u \in \mathbb{R}^d} f(Au)$ for any A: $|A| \neq$

(2) $\operatorname{argmin}_{x \in \mathbb{R}^d} f(x) = A\left(\operatorname{argmin}_{u \in \mathbb{R}^d} f(Au)\right)$ $\quad x = Au$

Set $g(u) = f(Au)$, say $f(w_1, w_2) = \frac{100}{2}w_1^2 + \frac{1}{2}w_2^2$

find an A to decrease K ($K = 100$)

$$A = \begin{bmatrix} \frac{1}{10} & 0 \\ 0 & 1 \end{bmatrix} \rightarrow g(u) = f(Au) = f\left(\frac{u_1}{10}, u_2\right)$$

$$= \frac{100}{2}\left(\frac{u_1}{10}\right)^2 + \frac{1}{2}u_2^2$$

$$= \frac{u_1^2}{2} + \frac{u_2^2}{2} \rightarrow K = 1?$$

$\frac{1}{\sqrt{\text{Hessian}}}$ $\quad K = 1$ is **MIN**

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

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

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 are just running gradient descent with gradients scaled by some positive semidefinite matrix $P = RR^T$.

This method is called **preconditioned** gradient descent, and we can apply the same idea to precondition SGD.

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

A <u>positive definite</u> $P \Rightarrow$ unique

TIME

	Preconditioned GD	Preconditioned SGD
time	base: $O(nd)$ $O(nd + d^2)$ ↑ multiply by P @ end	$O(d^2)$
space	$O(d^2)$	$O(d^2)$

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.

To address this, we use a DIAGONAL preconditioner: RESTRICT P to be a diagonal matrix.

TIME

	Preconditioned GD	Preconditioned SGD
time	base: $O(nd)$ $O(nd + d^2)$ ← multiply by P @ end	$O(d^2)$
space	$O(d^2)$	$O(d^2)$

How to Choose Preconditioner

(1) Use intuition about problem (Issue doesn't scale)

(2) Use dataset statistics

(3) Quasi-Newton methods → $R \approx (\nabla^2 f)^{-1/2}$, $P = (\nabla^2 f)^{-1}$ ✓ take diagonals → not always diagonal

Adaptive Learning Rates

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

$$(\omega_{t+1})_j = (\omega_t)_j - \alpha_{t,j}\left(\nabla f_{i_t}(\omega_t)\right)_j$$

where $(\omega_t)_j$ denotes the j^{th} entry of the model at time t, and the learning rate $\alpha_{t,j}$ is now:

(1) multi-parameter

(2) allowed to vary as a function of previously observed gradient samples.

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: AdaGrad

input: learning rate factor α, initial parameters ω.

initialize $r \leftarrow 0$

loop

sample a stochastic gradient $g \leftarrow \nabla f_{i_t}(\omega)$

accumulate second moment estimate $r_j \leftarrow r_j + g_j^2 \quad \forall j \in \{1,\dots,d\}$

update model $\omega_j \leftarrow \omega_j - \frac{\alpha}{\sqrt{r_j}} \cdot g_j$

end loop

Note: typically add small correction factor to avoid dividing by 0 if r_j is 0.

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. 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 small and vice versa.

Problem: It does not work well in non-convex setting, b/c learning rate depends on whole history of algorithm, and for non-convex optimization the trajectory may have passed through different 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. Modified 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.

A potential downside: step size does not generally go to zero, so could converge to a noise ball.

Algorithm RMSProp

input: learning rate factor α, decay rate ρ, initial parameters w.

initialize $r \leftarrow 0$

loop

Sample a stochastic gradient $g \leftarrow \nabla f_{i_t}(w)$

accumulate second moment estimate $r_j \leftarrow \rho r_j + (1-\rho) g_j^2$

update model $w_j \leftarrow w_j - \frac{\alpha}{\sqrt{r_j}} g_j$

end loop

Adam. Modified RMSProp to

(1) use momentum with exponential weighting

(2) correct for bias to estimate first & second order moments of the gradient.

Algorithm Adam

input: learning rate factor α, decay rates ρ_1, ρ_2, initial parameters w

initialize $r \leftarrow 0$, $s \leftarrow 0$

initialize timestep $t \leftarrow 0$

loop

$t \leftarrow t+1$

$g \leftarrow \nabla f_{i_t}(w)$ — sample a stochastic gradient

$s_j \leftarrow \rho_1 s_j + (1-\rho_1) g_j$ — accumulate first moment estimate

$r_j \leftarrow \rho_2 r_j + (1-\rho_2) g_j^2$ — accumulate second moment estimate

$\hat{s} \leftarrow s / (1-\rho_1^t)$ — correct first moment bias

$\hat{r} \leftarrow r / (1-\rho_2^t)$ — correct second moment bias

$w_j \leftarrow w_j - \frac{\alpha}{\sqrt{\hat{r}_j}} \hat{s}_j$ — update model

end loop

Why does Adam's bias correction work?

Want s to be a weighted avg of g.

i.e $s_t = \sum_{k=1}^{t} \gamma_k g_k$ such that $\sum_{k=1}^{t} \gamma_k = 1$

$$s_t = (1-\rho_1) g_t + \rho_1 (1-\rho_1) g_{t-1} + \cdots + \rho_1^{t-1} (1-\rho_1) g_1$$

$$= \sum_{k=0}^{t-1} \rho_1^k (1-\rho_1) = \frac{1-\rho_1^t}{1-\rho_1} (1-\rho_1) = 1-\rho_1^t$$

→ divide s_t by $1-\rho_1^t$ to get 1!