Gradient Descent

Preliminary (math stuff that will be useful)

1 Convexity

Definition 1.1. *A set C is said to be convex if for all $x, y \in C$ and $0 \leq t \leq 1$, the affine combination $(1-t)x + ty$ also belongs to C.*

Definition 1.2. *Let $f : \mathbb{R}^n \to \mathbb{R}$. For $0 \leq t \leq 1$ and $x, y \in \operatorname{dom}(f)$, if*

$$f(tx + (1-t)y) \leq tf(x) + (1-t)f(y)$$

then f is a convex function. If $f(tx + (1-t)y) < tf(x) + (1-t)f(y)$, then f is a strictly convex function. Note that f is convex if and only if its epigraph is a convex set.

Definition 1.3. *A matrix A is said to be positive semidefinite if and only if for any $x \in \mathbb{R}^n, x^T Ax \geq 0$. Similarly, A is positive definite if and only if $x^T Ax > 0$.*

For any matrix A, the *spectral* norm, or the induced 2-norm, is defined as

$$\|A\|_2 = \max_{x \neq 0} \frac{\|Ax\|_2}{\|x\|_2}$$

Definition 1.4. *Let A, B be any two matrices. Then, $A \succcurlyeq B$ means that for any vector x we have $x^T Ax \geq x^T Bx$. Notably, if $A \succcurlyeq 0$, then A is positive semidefinite.*

Theorem 1.1. *Let f be a convex function. Then, for any $x, y \in \operatorname{dom}(f)$,*

$$f(y) \geq f(x) + \nabla f(x) \cdot (y - x)$$

Furthermore, for any $x \in \operatorname{dom}(f)$,

$$\nabla^2 f(x) \succcurlyeq 0$$

Unrelated to convexity, but for any $f : \mathbb{R}^n \to \mathbb{R}$, the second order Taylor approximation is

$$f(x) = f(x_0) + (x - x_0)^T \nabla f(x_0) + \frac{1}{2}(x - x_0)^T \nabla^2 f(x_0)(x - x_0)$$

In most models, it is reasonable to assume that the gradient of the loss function is L-Lipschitz. Then, we have the following theorem.

Theorem 1.2. *Let f be a convex function such that $\nabla f(x)$ is L-Lipschitz. Then, $\nabla^2 f(x) \preccurlyeq L$, which means*

$$f(y) \leq f(x) + \nabla f(x) \cdot (y - x) + \frac{L}{2}\|x - y\|^2$$

What is GD?

- GD is x_{t + 1} = x_t - alpha * grad f(x_t).

- The idea of GD is simple – you are at a particular point, find a direction to move in that helps us get one step closer to the optimum (minimum or maximum), use that direction to make your final move (and now you're on a different spot).
- GD is about how we find that direction.

Rough intuition for how GD works

- The following theorem can be shown using some of the properties I gave above:

Theorem 6.1 *Suppose the function $f : \mathbb{R}^n \to \mathbb{R}$ is convex and differentiable, and that its gradient is Lipschitz continuous with constant $L > 0$, i.e. we have that $\|\nabla f(x) - \nabla f(y)\|_2 \leq L\|x - y\|_2$ for any x, y. Then if we run gradient descent for k iterations with a fixed step size $t \leq 1/L$, it will yield a solution $f^{(k)}$ which satisfies*

$$f(x^{(k)}) - f(x^*) \leq \frac{\|x^{(0)} - x^*\|_2^2}{2tk}, \tag{6.1}$$

where $f(x^*)$ is the optimal value. Intuitively, this means that gradient descent is guaranteed to converge and that it converges with rate $O(1/k)$.

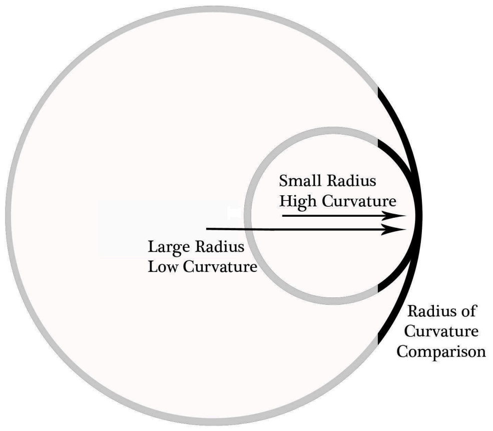
- Important takeaway here is that the convergence is O(1/k). It basically means that the f(x_k) - f(x*) follows the 1/k graph. Key thing to understand here is that increases in k in when k is small lead to significant decrease in the difference than when k is already large
- Below we connect gradient descent to curvature. This will set up discussion for preconditioning, adagrad, etc

$$\mathcal{L}(\theta_{t+1}) \leq \mathcal{L}(\theta_t) - \eta\|g_t\|^2 + \frac{1}{2}\eta^2\|g_t\|^2 L$$

Set $\eta = \frac{1}{L}$. Then

$$\mathcal{L}(\theta_{t+1}) \leq \mathcal{L}(\theta_t) - \frac{1}{L}\|g_t\|^2 + \frac{1}{2L}\|g_t\|^2 \implies \mathcal{L}(\theta_{t+1}) \leq \mathcal{L}(\theta_t) - \frac{\|g_t\|^2}{2L}$$

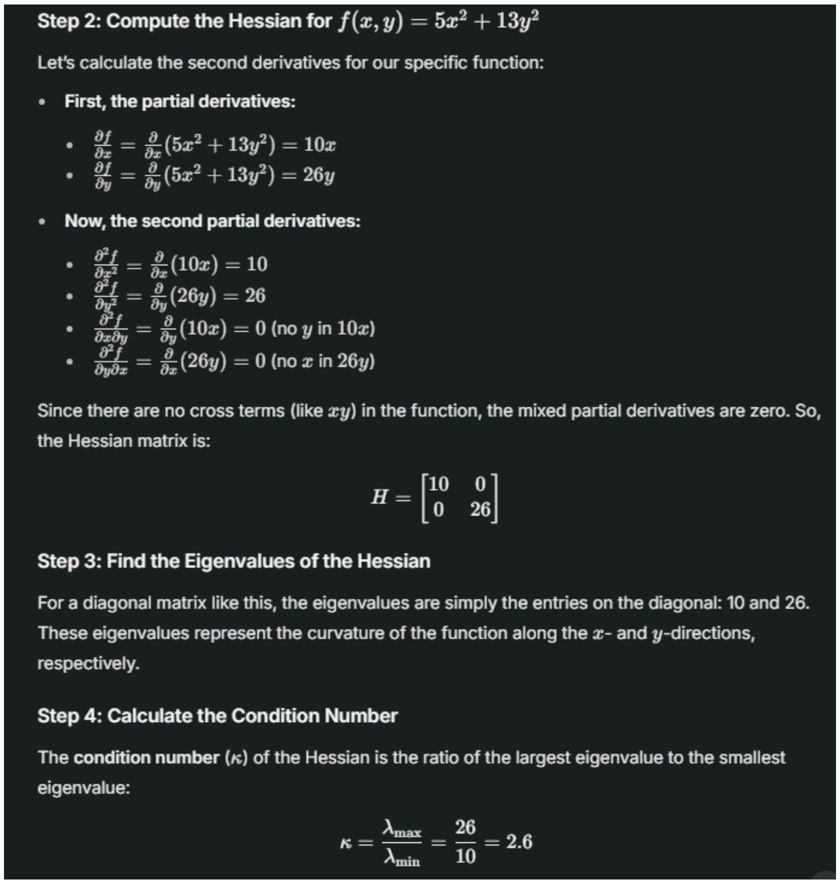
- This is just using some of the properties you saw above. Important is that last equation on the right side of the second line.
- Basically tells you that you want a large enough gradient relative to your curvature.
  - To review: curvature is how fast the slope is changing (it's the degree to which your function is "curved")
- Now, remember that you never subtract the whole gradient, like you always subtract the learning rate * the gradient.
  - And, so that's how you would "adjust" when you want more of the gradient and when you want less



- The problem is we never know what the local curvature (or the second gradient) is when we are doing gradient descent
- So, we will look into other ways of dealing with this

Preconditioning

- As we discussed earlier, curvature is key to gradient descent
- Now, we talked about one dimensional curvature (just using the second gradient). There is curvature along every dimension in higher dimensions. Furthermore, these can vary as well as you'd expect

  Hence, gradient descent becomes very difficult as certain dimensions require more "push" or "increment" than other dimensions. So, we must somehow account for all this
- When a function has curvature that is different along each dimension, we say that the function is ill-conditioned (we call this a *conditioning problem*)
- To determine whether a function is ill-conditioned, find it's Hessian matrix, then calculate the condition number of the matrix (the ratio of the largest eigenvalue to the smallest eigenvalue) → <u>in a sense, the eigenvalues tell you the dimensions of highest curvature and lowest curvature</u>
  - 

    Step 2: Compute the Hessian for $f(x,y) = 5x^2 + 13y^2$

    Let's calculate the second derivatives for our specific function:

    - **First, the partial derivatives:**
      - $\frac{\partial f}{\partial x} = \frac{\partial}{\partial x}(5x^2 + 13y^2) = 10x$
      - $\frac{\partial f}{\partial y} = \frac{\partial}{\partial y}(5x^2 + 13y^2) = 26y$
    - **Now, the second partial derivatives:**
      - $\frac{\partial^2 f}{\partial x^2} = \frac{\partial}{\partial x}(10x) = 10$
      - $\frac{\partial^2 f}{\partial y^2} = \frac{\partial}{\partial y}(26y) = 26$
      - $\frac{\partial^2 f}{\partial x \partial y} = \frac{\partial}{\partial y}(10x) = 0$ (no y in $10x$)
      - $\frac{\partial^2 f}{\partial y \partial x} = \frac{\partial}{\partial x}(26y) = 0$ (no x in $26y$)

    Since there are no cross terms (like xy) in the function, the mixed partial derivatives are zero. So, the Hessian matrix is:

    $$H = \begin{bmatrix} 10 & 0 \\ 0 & 26 \end{bmatrix}$$

    Step 3: Find the Eigenvalues of the Hessian

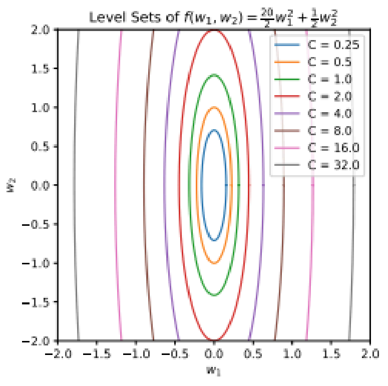
    For a diagonal matrix like this, the eigenvalues are simply the entries on the diagonal: 10 and 26. These eigenvalues represent the curvature of the function along the x- and y-directions, respectively.

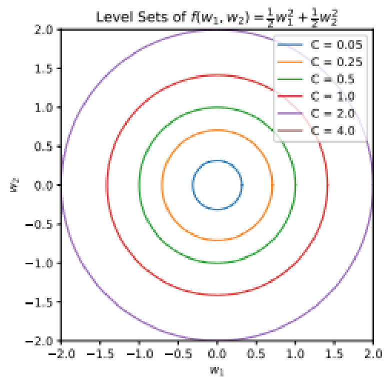
    Step 4: Calculate the Condition Number

    The **condition number** (κ) of the Hessian is the ratio of the largest eigenvalue to the smallest eigenvalue:

    $$\kappa = \frac{\lambda_{\max}}{\lambda_{\min}} = \frac{26}{10} = 2.6$$
  - Note that this only works for very simple examples. I attached one below. This is NOT to be taken as the norm (meaning that you can always do this). Just a proof of concept.

    - Another way to determine the condition number is to multiply the spectral norm of a matrix multiplied by the spectral norm of the inverse of the matrix
        - <u>Few things to remember here.</u>
        - What is the spectral norm? ||A|| = max ||Ax|| / ||x||. Basically, the 2-norm is a quantity that expresses how much a matrix (which is a transformation) can stretch a vector.
            - When we use "||" here, if it's a vector, then it's just the normal euclidean norm.
        - Now, if you replace the max with a min, then actually min ||Ax|| / ||x|| = 1 / ||A^{-1}|| (where this is actually the spectral norm with max).
        - This definition also better helps put into perspective the idea that the condition number of the hessian reflects how much the original function can stretch or shrink a particular vector
        - This is the reason why when you have ill-conditioned functions, gradient descent ends up being zig-zagged, because one dimension is being pulled way more than the other, and so you end up being poorly aligned, and so you "flip-flop"
        - Condition number = 1 → perfectly conditioned. If it's greater than 1, then it gets worse and worse.
- Now, how does it work?





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

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

- Idea is you know you have something that is ill-conditioned, so you try to fix it by considering transforming the entire space (that's what R is doing), and that's what g(u) = f(Ru) is. It's basically trying to fix this ill-conditioned space.

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.

- So, they take what we did above, multiply it by that same R, and get w_t - alpha * RR^T * gradient. Then, it seems like you ignore w_t, and just focus on the RR^T outside of the gradient. This is what's called a **preconditioner**. Basically, you take any positive semidefinite matrix, and use it as a preconditioner.
- Okay so how can we find a preconditioner or what can we use?
    - Information about the loss that you know already. This means we set the preconditioner to what we want. This is a static choice, so it just works for one function, probably "easy" too (toy examples)
    - Statistics of the dataset can be another preconditioner. Variance of the features in your dataset. Maybe you can have Var(X_i), and then have a diagonal matrix.
    - Information about the second derivative. This is what we will discuss next. You can think of these as "Hessian preconditioners"

Hessian Preconditioners

- Idea is to somehow use the hessian matrix as a preconditioner.

The second-order Taylor expansion of a function $f(x)$ around a point x_t is given by

$$f(x) \approx f(x_t) + \langle \nabla f(x_t), x - x_t \rangle + \frac{1}{2}\langle x - x_t, \nabla^2 f(x_t)(x - x_t)\rangle$$

where $\nabla^2 f(x_t)$ is the Hessian matrix of f at x_t. The minimum of $f(x)$ can be found in closed form by setting the gradient (with respect to x) of the above expression to zero, which gives

$$\nabla f(x_t) + \nabla^2 f(x_t)(x - x_t) = 0 \qquad \Longrightarrow \qquad x = x_t - [\nabla^2 f(x_t)]^{-1} \nabla f(x_t).$$

So, we find that by moving from first-order to second-order Taylor approximation, and assuming that $\nabla^2 f(x_t)$ is invertible, the natural direction of descent changes from

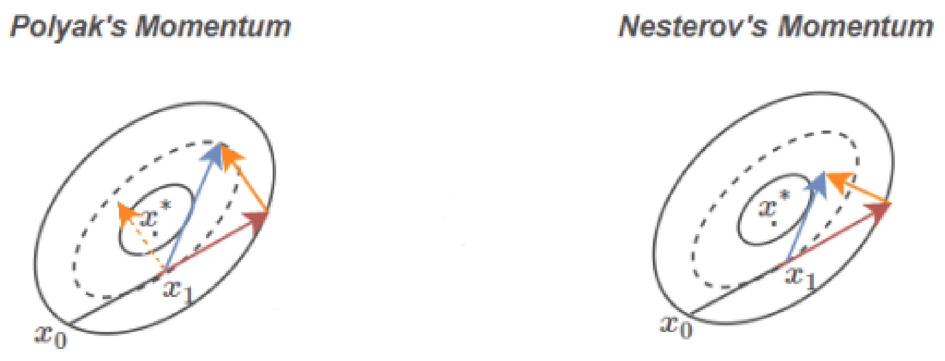
$$\underbrace{d = -\nabla f(x_t)}_{\substack{\text{using first-order} \\ \text{Taylor approximation}}} \qquad \text{to} \qquad \underbrace{d = -[\nabla^2 f(x_t)]^{-1} \nabla f(x_t)}_{\text{using second-order Taylor approximation}} .$$

- From the first line to the second line, when they take the gradient, gradient of f(x_t) = 0, gradient of grad f(x_t) dot (x - x_t) is grad f(x_t), and gradient of 1/2 * <x - x_t, hessian f(x_t) * (x - x_t)> is hessian f(x_t) * (x - x_t). That's how they get that term on the left for the second line.
- Then you solve, and so you get x_t - inverse of hessian f(x_t) * grad f(x_t).
- This is probably the most effective way to tackle the conditioning problem. The reason is because the inverse of the hessian matrix will have the inverse eigenvalue across every single dimension, and so you perfectly "scale" the gradients by doing this type of update (as you see in the image above)

- A big issue though is computationally and memory wise - computing a hessian matrix is really difficult when you have millions of parameters. Hessian matrices are order N^2 where N is the number of parameters

Momentum

- Momentum is based on a principle we discussed earlier about curvature:
    - Areas of high curvature → let's take smaller steps
    - Areas of lower curvature → let's take larger steps
- This image is really nice to visualize how it works.
- Polyak momentum is actually just an EMA of gradients that have happened up till time stamp t.
    - That is: m_{t + 1} = gamma * m_{t} + (1 - gamma) * gradient of f(x_t), then x_{t + 1} = x_{t} - alpha * m_{t + 1}. It just can be rewritten into what you see below



$$x_{t+1} = x_t - \alpha \nabla f(x_t) + \mu(x_t - x_{t-1})$$

$$x_{t+1} = x_t + \mu(x_t - x_{t-1}) - \gamma \nabla f(x_t + \mu(x_t - x_{t-1})$$

- Basically, the idea of polyak momentum is to reduce the number of oscillations that happen when we are in areas of high curvature. [This link talks has visuals if you need.]
- Nesterov is just an improvement upon this.
- Polyak is O(1/t) convergence, Nesterov is O(1/t^2) convergence. So, Nesterov is significantly faster.

AdaGrad

The general AdaGrad update rule is given by:

$$x_{t+1} = x_t - \eta G_t^{-1/2} g_t$$

where $G_t^{-1/2}$ is the inverse of the square root of G_t. A simplified version of the update rule takes the diagonal elements of G_t instead of the whole matrix:

$$x_{t+1} = x_t - \eta \mathrm{diag}(G_t)^{-1/2} g_t$$

which can be computed in linear time. In practice, a small quantity ϵ is added to each diagonal element in G_t to avoid singularity problems, the resulting update rule is given by:

$$x_{t+1} = x_t - \eta \mathrm{diag}(\epsilon I + G_t)^{-1/2} g_t$$

where I denotes the identity matrix. An expanded form of the previous update is presented below,

$$\begin{bmatrix} x_{t+1}^{(2)} \\ x_{t+1}^{(2)} \\ \vdots \\ x_{t+1}^{(m)} \end{bmatrix} = \begin{bmatrix} x_t^{(2)} \\ x_t^{(2)} \\ \vdots \\ x_t^{(m)} \end{bmatrix} - \begin{bmatrix} \eta \frac{1}{\sqrt{\epsilon + G_t^{(1,1)}}} \\ \eta \frac{1}{\sqrt{\epsilon + G_t^{(2,2)}}} \\ \vdots \\ \eta \frac{1}{\sqrt{\epsilon + G_t^{(m,m)}}} \end{bmatrix} \odot \begin{bmatrix} g_t^{(2)} \\ g_t^{(2)} \\ \vdots \\ g_t^{(m)} \end{bmatrix}$$

- AdaGrad stands for Adaptive Gradient.
- Here, G_t refers to sum_{tau = 1}^t g_{tau} * g_{tau}^T (so remember G_t) is a large matrix with dimension of N^2 (where N is the dimension of the gradient)
- Now, as you can imagine (and we discussed earlier), this is bad for memory, so we sometimes just take the diagonal of the matrix, and so that's what you see at the very end
    - Also, remember that the inverse of a diagonal matrix is the reciprocal entry in the diagonal. So, everything should make sense here
- Notice how Adagrad is just another preconditioning matrix as we discussed earlier
    - Basically, it's just accumulating past gradients, and then making an assumption that if you have more gradient along one dimension, so then don't apply as much to that one dimension (that's why it's 1 / sqrt of accumulated gradient)
- A key thing to note is that the convergence bound for Adagrad is 1/sqrt(t). Nice way to remember this is that you have a square root of accumulated gradient. 1/sqrt(t) is slower than before though
    - However, idea is that Adagrad works better in ill-conditioned environments while normal gradient descent doesn't really work in those environments

RMSProp

- RMSProp is Adagrad, but you do an EMA approximation for the squared mean of the gradients (which in other words, is also the second moment of the gradients around 0)
- This is what RMSProp looks like:
    - v_t = beta * v_{t - 1} + (1 - beta) * g_t^2
    - theta_{t + 1} = theta_t - learning rate / sqrt(v_t + epsilon) * g_t
- Again, it's the preconditioning idea, but notice it's literally the same thing as Adagrad, but we use an EMA instead of accumulating
- Why RMSProp? What issue does it solve?
    - Adagrad has an issue where the accumulating gradient becomes large, leading to a vanishing gradient update
    - Hence, rather than accumulating, we take the average. This is much more stable.

- In my opinion, it’s a very small improvement, but it makes sense why they did it though

ADAM

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$$\theta_t = \theta_{t-1} - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

- Okay let’s ignore the third and fourth equation. It’s basically the same thing as RMSProp, but we include momentum’s EMA as well for the gradient update.
- Now, the third and fourth equation is called the bias correction. Look at this below ChatGPT response:

Concretely, if g_i^2 has a mean σ^2, then

$$E[v_t] = E\left[(1-\beta_2)\sum_{i=1}^{t} \beta_2^{t-i} g_i^2\right] = (1-\beta_2)\sum_{i=1}^{t} \beta_2^{t-i} E[g_i^2] = \sigma^2 (1-\beta_2)\sum_{i=1}^{t} \beta_2^{t-i}.$$

Since $\sum_{i=1}^{t} \beta_2^{t-i} = \sum_{k=0}^{t-1} \beta_2^k = \frac{1-\beta_2^t}{1-\beta_2}$, we get

$$E[v_t] = \sigma^2 (1-\beta_2) \frac{1-\beta_2^t}{1-\beta_2} = \sigma^2 \left(1-\beta_2^t\right).$$

Notice that for small t, $1-\beta_2^t \approx 0$. So $E[v_t]$ is **biased low** in the early steps (it starts at zero and ramps up).

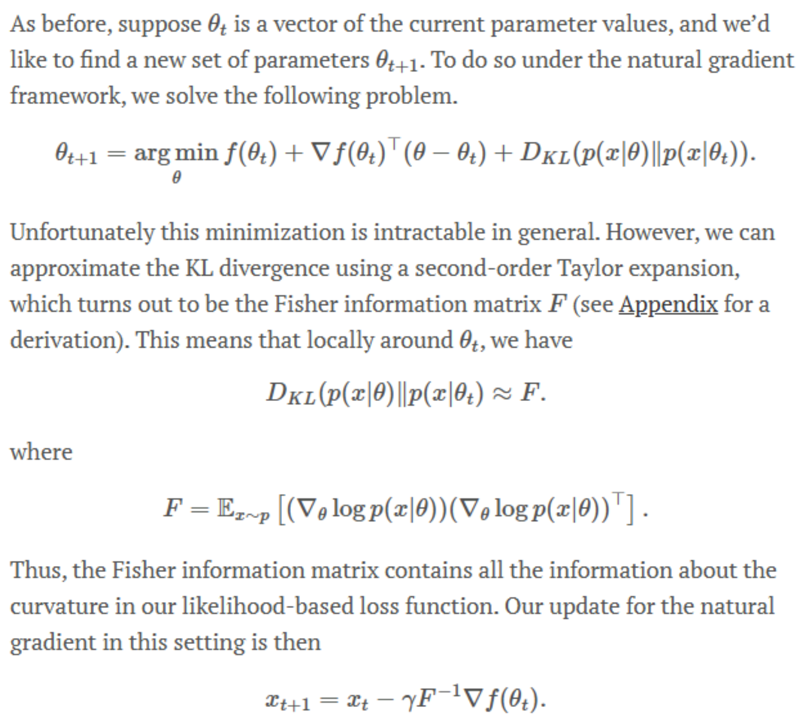
- So, one general thing to note is that EMAs are underestimates for the first/second moment.

Natural Gradient Descent

- NGD is just gradient descent but generalized to different “spaces”. We have seen gradient descent being used to optimize functions, but as we’ll see in reinforcement learning, gradient descent is also used to optimize probability distributions

$$x_{t+1} = \arg\min_x f(x_t) + \nabla f(x_t)^\top (x - x_t) + D(x, x_t)$$

- For natural gradient descent, D(x, x_t) = ½ * ||x - x_t||, so gradient descent on the euclidean space
    - To see why, take the gradient of f(x_t) + grad f(x_t) * (x - x_t) + ½ ||x - x_t||
    - This is grad f(x_t) + x - x_t = 0. Solving, you get x = x_t - grad f(x_t), so x_{t + 1} = x_t - grad f(x)
    - And, this is the typical gradient descent computation!
    - So, gradient descent is basically the optimal value of the first order approximation of a function



As before, suppose θ_t is a vector of the current parameter values, and we'd like to find a new set of parameters θ_{t+1}. To do so under the natural gradient framework, we solve the following problem.

$$\theta_{t+1} = \arg\min_{\theta} f(\theta_t) + \nabla f(\theta_t)^\top (\theta - \theta_t) + D_{KL}(p(x|\theta) \| p(x|\theta_t)).$$

Unfortunately this minimization is intractable in general. However, we can approximate the KL divergence using a second-order Taylor expansion, which turns out to be the Fisher information matrix F (see Appendix for a derivation). This means that locally around θ_t, we have

$$D_{KL}(p(x|\theta) \| p(x|\theta_t) \approx F.$$

where

$$F = \mathbb{E}_{x \sim p}\left[(\nabla_\theta \log p(x|\theta))(\nabla_\theta \log p(x|\theta))^\top\right].$$

Thus, the Fisher information matrix contains all the information about the curvature in our likelihood-based loss function. Our update for the natural gradient in this setting is then

$$x_{t+1} = x_t - \gamma F^{-1} \nabla f(\theta_t).$$

- Some other D(x, x_t):
    - Mahalanobis metric → D(x, x_t) = ½ xAx (A is the matrix you pick)
        - This is actually similar to gradient descent with preconditioning matrix
    - Fisher information metric: D(theta, theta_t) = D_{kl}(p(x | theta), p(x | theta_t))
        - So, the distance between two distributions. Theta_t is the previous, theta is the one that we are trying to pick
- For the fisher information metric, the KL divergence between the two distributions is usually approximated as the fisher information of p(x | theta).
    - So, the gradient descent is update is just F^{-1} (where F is the fisher information matrix) as a preconditioner

Fisher information approximates the KL divergence

For notational simplicity, let $D(\theta, \theta_t) = D_{KL}(p_\theta(x) | p_{\theta_t}(x))$. Consider a second-order Taylor approximation to the KL divergence around θ_t,

$$D(\theta, \theta_t) \approx D(\theta_t, \theta_t) + \left(\nabla_\theta D(\theta, \theta_t)\big|_{\theta=\theta_t}\right)^\top (\theta - \theta_t) + (\theta - \theta_t)^\top H_t(\theta - \theta_t)$$

where H_t is the Hessian of $D(\theta_t, \theta_t)$ at θ_t.

The first two terms are zero. The first term is a divergence between two equal distributions, which makes the divergence zero. For the second term, we can see that

$$\begin{aligned}
\nabla_\theta D(\theta, \theta_t) &= \nabla_\theta \mathbb{E}_{p(x|\theta)}\left[\log \frac{p(x|\theta)}{p(x|\theta_t)}\right] \\
&= \mathbb{E}_{p(x|\theta)}\left[\nabla_\theta \log \frac{p(x|\theta)}{p(x|\theta_t)}\right] && \text{(Swap } \nabla \text{ and } \mathbb{E}\text{)} \\
&= \mathbb{E}_{p(x|\theta)}\left[\nabla_\theta \log p(x|\theta)\right] && \text{(Grad. doesn't depend on } \theta_t\text{)} \\
&= 0.
\end{aligned}$$

The final line comes from the fact that the expectation of the score is always 0.