**Polyak Step Size**

* **Polyak choice of step size:** \( \alpha = \frac{f(x) - f^*}{\nabla f(x)^T \nabla f(x)} \)
  with \( \nabla f(x)^T \nabla f(x) = \sum_{i=1}^{n} \frac{\partial f}{\partial x_i}(x)^2 \) and \( f^* = \min_x f(x) \)

* Where does this come from? Recall \( f(x + \delta) \approx f(x) + \nabla f(x)^T \delta \). Choosing \( \delta = -\alpha \nabla f(x) \) then

  \[
  f(x - \alpha \nabla f(x)) \approx f(x) - \alpha \nabla f(x)^T \nabla f(x)
  \]

* Choosing \( \alpha = \frac{f(x) - f^*}{\nabla f(x)^T \nabla f(x)} \) then

  \[
  f(x - \alpha \nabla f(x)) \approx f(x) - \frac{(f(x) - f^*)}{\nabla f(x)^T \nabla f(x)} \nabla f(x)^T \nabla f(x) = f^*
  \]

* Scaling step size with \( \frac{1}{\nabla f(x)^T \nabla f(x)} \) makes lots of sense → when gradient is large the step size is small, when the gradient is small the step size is large.

* Don’t really need to know \( f^* \), can estimate it or just using \( f^* = 0 \) can work well in ML problems.

* Much lower computational burden than line search ...

---

1 Originally proposed in Polyak’s book, for more recent work see e.g. https://arxiv.org/pdf/1905.00313.pdf
Polyak Step Size

* Polyak choice of step size: \( \alpha = \frac{f(x) - f^*}{\nabla f(x)^T \nabla f(x)} \)

* When close to minimum then expect \( \nabla f(x) \) to be close to zero, and so \( \nabla f(x)^T \nabla f(x) \approx 0 \). To avoid division by zero, in practice tend to modify Polyak step size to

\[
\alpha = \frac{f(x) - f^*}{\nabla f(x)^T \nabla f(x) + \epsilon}
\]

where \( \epsilon \) is a small number e.g. 0.001

* Example: \( f = x^2 \), starting point \( x = 1 \), \( f^* = 0 \), \( \epsilon = 0.0001 \) and \( \epsilon = 0 \)

* With \( \epsilon = 0 \) Polyak step size is constant, with \( \epsilon = 0.0001 \) Polyak step decreases as get close to minimum. Why?
* Example: $f = x^2$, starting point $x = 1$, $f^* = 0$, $\epsilon = 0.0001$ and $\epsilon = 0$

\[ \frac{df}{dx} = 2x \rightarrow \text{Polyak choice is } \alpha = \frac{f(x) - f^*}{\nabla f(x)^T \nabla f(x) + \epsilon} = \frac{x^2 - 0}{(2x)^2 + \epsilon} = \frac{x^2}{4x^2 + \epsilon} \]

When $\epsilon = 0$, $\alpha = \frac{x^2}{4x^2} = 0.25 \rightarrow$ constant

When $\epsilon = 0.0001$ then when $x^2 \approx 0$ close to minimum $\alpha \approx \frac{x^2}{\epsilon} \approx 0$.

* Using non-zero $\epsilon$ affects behaviour close to minimum, but effect is small if $\epsilon$ is small (remember in ML we usually don’t need to get very close to the minimum, just close to the noise floor).
Suppose we didn’t know \( f^* \) exactly but used a lower value. NB: Shouldn’t use an \( f^* \) larger than true min coz then \( f(x) - f^* < 0 \) and we might end up with \( \alpha < 0 \).

Example: \( f = x^2 + 0.001 \), min is now \( f(0) = 0.001 \) when \( x = 0 \). Starting point \( x = 1 \), Polyak \( f^* = 0 \), \( \epsilon = 0.0001 \) and \( \epsilon = 0.00001 \).

Polyak converges quickly, but then step size becomes large and “noisy”

When \( \epsilon = 0 \), \( \alpha = \frac{x^2 + 0.001}{4x^2} \). When \( x^2 \approx 0 \) then \( \alpha = \frac{0.001}{4x^2} \) so badly behaved

When \( \epsilon = 0.0001 \) then when \( x^2 \approx 0 \) then \( \alpha \approx \frac{x^2 + 0.001}{\epsilon} \approx \frac{0.001}{0.0001} = 10 \), so too large

 Might choose to place upper limit on step size?
Examples

Polyak $\epsilon = 0.0001$, $f^* = 0$

* Quadratic:

* Quadratic loss:

* Since added computational burden of Polyak is low, its competitive with line search wrt convergence vs wall-clock time
Examples
Polyak $\epsilon = 0.0001, f^* = 0$

* Rosenbrock function:

* Toy neural net loss:

* Again, Polyak competitive with line search wrt convergence vs wall-clock time

* Hasn’t been much work looking at Polyak for ML
Polyak $\epsilon = 0.0001$, $f^* = 0$

- Non-smooth function $f(x) = |x_1| + x_2^2$

![Graph showing function values over iterations and time](image)

- Note: Polyak hasn’t received much attention for ML → but ML usually uses SGD, will come on to this shortly
Adagrad\(^2\)

* Another idea for adapting step size:

\[
\begin{align*}
\mathbf{x}_0 &= \mathbf{x}0; \alpha_0 = \alpha 0; \text{sum} = \epsilon; t = 0 \\
\text{for } k \text{ in range(num\_iters):} \\
\mathbf{x}_{t+1} &= \mathbf{x}_t - \alpha_t \frac{df}{dx}(\mathbf{x}_t) \\
\text{sum} &= \text{sum} + \frac{df}{dx}(\mathbf{x}_t)^2 \\
\alpha_{t+1} &= \frac{\alpha 0}{\text{sum}} \\
t &= t + 1
\end{align*}
\]

i.e. step size at iteration \(t\) is:

\[
\alpha_t = \frac{\alpha_0}{\sqrt{\frac{df}{dx}(\mathbf{x}_0)^2 + \frac{df}{dx}(\mathbf{x}_1)^2 + \cdots + \frac{df}{dx}(\mathbf{x}_{t-1})^2 + \epsilon}} = \frac{\alpha_0}{\sqrt{\sum_{i=1}^{t-1} \frac{df}{dx}(\mathbf{x}_i)^2 + \epsilon}}
\]

* Similar to Polyak, but now divide by sum of gradients rather than just by last gradient. Still have to manually select \(\alpha_0\).

* Idea: (i) step size should get smaller over time (why?) and (ii) should be larger/smaller depending on past gradients.

\(^2\)https://www.jmlr.org/papers/volume12/duchi11a/duchi11a.pdf, see Section 1.3.1 in particular. Quite mathematical ...
Adagrad

* In general, $x$ is a vector and $\nabla f(x_t) = \left[ \frac{\partial f}{\partial x_1}(x), \frac{\partial f}{\partial x_2}(x), \ldots, \frac{\partial f}{\partial x_n}(x) \right]$ is a vector. So far we’ve used a single step size:

$$x_{t+1} = x_t - \alpha \left[ \frac{\partial f}{\partial x_1}(x), \frac{\partial f}{\partial x_2}(x), \ldots, \frac{\partial f}{\partial x_n}(x) \right]$$

* But Adagrad uses a vector of step sizes $[\alpha_1, \alpha_2, \ldots, \alpha_n]$ and:

$$x_{t+1} = x_t - [\alpha_1 \frac{\partial f}{\partial x_1}(x), \alpha_1 \frac{\partial f}{\partial x_2}(x), \ldots, \alpha_n \frac{\partial f}{\partial x_n}(x)]$$

with

$$\alpha_1 = \frac{\alpha_0}{\sqrt{\sum_{i=1}^{t-1} \left( \frac{\partial f}{\partial x_1}(x_i) \right)^2 + \epsilon}}$$

$$\alpha_2 = \frac{\alpha_0}{\sqrt{\sum_{i=1}^{t-1} \left( \frac{\partial f}{\partial x_2}(x_i) \right)^2 + \epsilon}}$$

etc
Example: $f = x^2$, starting point $x = 1$, Adagrad $\alpha_0 = 0.2$ (selected so that initial step size is 0.1, see below), $\epsilon = 1.0e - 5$, constant $\alpha = 0.1$

Adagrad and constant step size initially the same. Adagrad decreases step size, quickly at first (when $df/dx$ is larger) then much more slowly as close to minimum $df/dx$ is small.
* Adjust $\alpha_0$ for Adagrad so that later step size matches that constant step size strategy ...

* Example: $f = x^2$, starting point $x = 1$, Adagrad $\alpha_0 = 0.3$ (larger now), $\epsilon = 1.0e - 5$, constant $\alpha = 0.1$
Examples

* Quadratic: Adagrad $\alpha_0 = 0.5$, $\epsilon = 1.0e - 5$ (note larger value of $\alpha_0$), constant $\alpha = 0.1$

![Graph 1](image1.png)

![Graph 2](image2.png)

* Quadratic loss: Adagrad $\alpha_0 = 2.5$, $\epsilon = 1.0e - 5$ (note huge value of $\alpha_0$), constant $\alpha = 0.5$

![Graph 3](image3.png)

![Graph 4](image4.png)

* Important to tune Adagrad initial step size $\alpha_0$, but with tuning can get fast convergence.
Examples

* Rosenbrock function: Adagrad $\alpha_0 = 1, \epsilon = 1.0e - 5$, constant $\alpha = 0.002$

* Toy neural net loss: Adagrad $\alpha_0 = 2.5, \epsilon = 1.0e - 5$, constant $\alpha = 0.5$

* Even with tuning $\alpha_0$ Adagrad isn’t faster than constant step size strategy on Rosenbrock function. But fast convergence for neural net loss function.
Examples

Adagrad $\alpha_0 = 0.01, \epsilon = 1.0e-5$, constant $\alpha = 0.005$

* Non-smooth function $f(x) = |x_1| + x_2^2$

* Recall that kink in cost function tends to cause “chattering” when use constant step size close to minimum, by decreasing the step size Adagrad reduces this.
RMSprop

* Adagrad can decrease the step size too aggressively (that’s why we needed to make $\alpha_0$ large), making convergence slow. RMSprop is a tweak to try to address that.

* Adagrad:

$$
\alpha_t = \frac{\alpha_0}{\sqrt{\frac{df}{dx}(x_0)^2 + \frac{df}{dx}(x_1)^2 + \cdots + \frac{df}{dx}(x_{t-1})^2 + \epsilon}}
$$

* RMSprop: Gradually forget past values:

$$
\alpha_t = \frac{\alpha_0}{\sqrt{(1 - \beta) \beta^t \frac{df}{dx}(x_0)^2 + (1 - \beta) \beta^{t-1} \frac{df}{dx}(x_1)^2 + \cdots + (1 - \beta) \frac{df}{dx}(x_{t-1})^2 + \epsilon}}
$$

with $0 < \beta \leq 1$. Implement using:

```
x_0 = x0; \alpha_0 = \alpha0; sum = 0; t = 0

for k in range(num_iters):
    x_{t+1} = x_t - \alpha_t \frac{df}{dx}(x_t)
    sum = \beta sum + (1 - \beta) \frac{df}{dx}(x_t)^2
    \alpha_{t+1} = alpha0/(\sqrt{sum} + \epsilon)
    t = t + 1
```

* Have to manually select $\alpha_0$ and $\beta$. 

Proposed in Geoff Hinton lecture [link]
\[ \alpha_t = \frac{\alpha_0}{\sqrt{(1 - \beta)(\beta^t \frac{df}{dx}(x_0)^2 + \beta^{t-1} \frac{df}{dx}(x_1)^2 + \cdots + \frac{df}{dx}(x_{t-1})^2) + \epsilon}} \]

Implement denominator using:

\[ \text{sum} = \beta \text{sum} + (1 - \beta) \frac{df}{dx}(x_t)^2 \]

and then use \( \sqrt{\text{sum}} + \epsilon \)

* How to choose \( \beta \)?

* Typically choose \( \beta = 0.9 \) or larger. For small \( \beta \) RMSprop reverts to Adagrad.
Example: $f = x^2$, starting point $x = 1$, RMSprop $\alpha_0 = 0.06$, $\beta = 0.9$, Adagrad $\alpha_0 = 0.2$, $\epsilon = 1.0e-5$, constant $\alpha = 0.1$

Adagrad, RMSprop and constant step size initially the same. RMSprop eventually increases step size - why?
Examples

- Quadratic: RMSprop $\alpha_0 = 0.15$, $\beta = 0.9$, $\epsilon = 1.0e-5$, Adagrad $\alpha_0 = 0.5$, constant $\alpha = 0.1$

- Quadratic loss: RMSprop $\alpha_0 = 0.5$, $\beta = 0.9$, $\epsilon = 1.0e-5$, Adagrad $\alpha_0 = 2.5$, constant $\alpha = 0.5$

- Adagrad, RMSprop, Polyak all pretty similar. Note need to tune $\alpha_0$ for Adagrad, RMSprop.
Examples

* Rosenbrock function: RMSprop $\alpha_0 = 0.01$, $\beta = 0.9$, $\epsilon = 1.0e - 5$, Adagrad $\alpha_0 = 1$, constant $\alpha = 0.002$

![Graphs of function values and iterations for the Rosenbrock function with different optimization methods.]

* Toy neural net loss: RMSprop $\alpha_0 = 0.05$, $\beta = 0.9$, $\epsilon = 1.0e - 5$, Adagrad $\alpha_0 = 1$, constant $\alpha = 0.75$

![Graphs of function values and iterations for the toy neural net loss function with different optimization methods.]

* Even with tuning $\alpha_0$ RMSprop/Adagrad are not faster than constant step size strategy on Rosenbrock function. But fast convergence for neural net loss function.
RMSprop $\alpha_0 = 0.01$, $\beta = 0.9$, $\epsilon = 1.0 \times 10^{-5}$, Adagrad $\alpha_0 = 0.01$, constant $\alpha = 0.005$

- Non-smooth function $f(x) = |x_1| + x_2^2$

- RMSprop less effective than Adagrad at reducing chattering close to minimum of non-smooth function (doesn’t reduce step size enough).
Per co-ordinate step size

* Adagrad/RMSprop use a vector of step sizes $[\alpha_1, \alpha_2, \ldots, \alpha_n]$ and:

$$x_{t+1} = x_t - [\alpha_1 \frac{\partial f}{\partial x_1}(x), \alpha_1 \frac{\partial f}{\partial x_2}(x), \ldots, \alpha_n \frac{\partial f}{\partial x_n}(x)]$$

* Other classes of approaches that also use a vector $\alpha$ include *Newton* methods and *quasi-Newton* methods.

* Newton methods use the second-derivative (the derivative of the derivative) to calculate $\alpha$. For $\nabla f(x_t) = [\frac{\partial f}{\partial x_1}(x), \frac{\partial f}{\partial x_2}(x), \ldots, \frac{\partial f}{\partial x_n}(x)]$ then second derivative is a matrix (a 2D array):

$$
\begin{pmatrix}
\frac{\partial}{\partial x_1} \frac{\partial f}{\partial x_1}(x) & \frac{\partial}{\partial x_1} \frac{\partial f}{\partial x_2}(x) & \cdots & \frac{\partial}{\partial x_1} \frac{\partial f}{\partial x_n}(x) \\
\frac{\partial}{\partial x_2} \frac{\partial f}{\partial x_1}(x) & \frac{\partial}{\partial x_2} \frac{\partial f}{\partial x_2}(x) & \cdots & \frac{\partial}{\partial x_2} \frac{\partial f}{\partial x_n}(x) \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial}{\partial x_n} \frac{\partial f}{\partial x_1}(x) & \frac{\partial}{\partial x_n} \frac{\partial f}{\partial x_2}(x) & \cdots & \frac{\partial}{\partial x_n} \frac{\partial f}{\partial x_n}(x)
\end{pmatrix}
$$

* Expensive to calculate. Quasi-Newton methods (e.g. *BFGS*) try to approx second derivative more efficiently, but are still computationally costly $\rightarrow$ only suitable for “small” data. Newton/quasi-Newton methods can accelerate convergence vs #iterations but can be slower wrt wall-clock time

* Can perform poorly on non-convex problems (may not converge at all e.g. see Fig 1 in https://arxiv.org/pdf/1606.01885v1.pdf, https://epubs.siam.org/doi/10.1137/S1052623401383455)
Polyak choice of step size works pretty well in examples here, but not well studied for ML problems.

Adagrad is known to suffer from slow convergence due to decreasing step size too quickly. RMSprop tries to patch up Adagrad. Sometimes helps, sometimes doesn’t in examples here.

Adagrad and RMSprop are implemented in common ML libraries, but Polyak isn’t … hmm?

In ML its normal to use SGD, we’ll come to this later.