AI505

Optimization

# First-Order Methods

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

# Outline

# Descent direction methods

How to select the descent direction?

- first-order methods that rely on gradient

- second-order methods that rely on Hessian information

cheap iterations good for small and lage scale optimization embedded optimziation
helpful because easy to warm restart
Limitations
not hard to find challenging instances for them.
can converge slowly.
limited computational power.

# Gradient Descent

The **steepest descent** direction at $x_k$, at $k$th iteration of a **local descent iterative method**, is the one opposite to the gradient (**gradient descent**):

$$\boldsymbol{g}_k = \nabla f(\boldsymbol{x}_k)$$
$$\boldsymbol{d}_k = -\frac{\boldsymbol{g}_k}{||\boldsymbol{g}_k||}$$

Guaranteed to lead to improvement if:

- $f$ is smooth
- step size is sufficiently small
- $x_k$ is not a stationary point (ie, $\nabla f(x_k) = 0$)

# Gradient Descent: Example

- Suppose we have

$$f(\boldsymbol{x}) = x_1 x_2^2$$

- The gradient is $\nabla f = [x_2^2, 2x_1 x_2]$
- $x_k = [1, 2]$

$$\boldsymbol{d}_{k+1} = -\frac{\nabla f(\boldsymbol{x}_k)}{\|\nabla f(\boldsymbol{x}_k)\|} = \frac{[-4, -4]}{\sqrt{16+16}} = \left[-\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}\right]$$

```
class DescentMethod:
    alpha: float

class GradientDescent(DescentMethod):
    __init__(self, f, grad, x, alpha):
        self.alpha = alpha

    def step(self, f, grad, x):
        alpha, g = self.alpha, grad(x)
        return x - alpha * g
```

# Gradient Descent

Th.: The next direction is orthogonal to the current direction.

Proof:

$$\alpha_k^* = \underset{\alpha}{\text{argmin}} f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$$

$$\nabla f(\boldsymbol{x}_k + \alpha_k^* \boldsymbol{d}_k) = \nabla_{\boldsymbol{d}_k} f(\boldsymbol{x}_k) = 0 \qquad \text{because } \alpha_k^* \text{ is miminum}$$

$$\nabla f(\boldsymbol{x}_k + \alpha_k^* \boldsymbol{d}_k)^T \boldsymbol{d}_k = 0 \qquad \text{because directional derivative: } \nabla_{\boldsymbol{s}} f(\boldsymbol{x}) = \nabla f(\boldsymbol{x})^T \boldsymbol{s}$$

$$\boldsymbol{d}_{k+1} = -\frac{\nabla f(\boldsymbol{x}_k + \alpha_k^* \boldsymbol{d}_k)}{\|\nabla f(\boldsymbol{x}_k + \alpha_k^* \boldsymbol{d}_k)\|} \qquad \text{gradient descent}$$

$$\boldsymbol{d}_{k+1} \cdot \boldsymbol{d}_k = -\frac{\nabla f(\boldsymbol{x}_k + \alpha_k^* \boldsymbol{d}_k)}{\|\nabla f(\boldsymbol{x}_k + \alpha_k^* \boldsymbol{d}_k)\|} \cdot \boldsymbol{d}_k = 0 \qquad \boldsymbol{d}_{k+1}^T \boldsymbol{d}_k = 0 \implies \boldsymbol{d}_{k+1} \perp \boldsymbol{d}_k$$

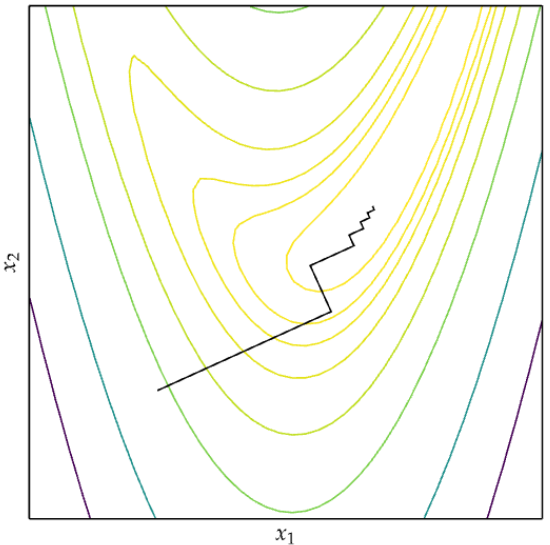
□

# Gradient Descent: Example

2D Rosenbrock function

$$f(x, y) = (a - x)^2 + b(y - x^2)^2$$

Narrow valleys not alligned with gradient can be a problem



# Conjugate Gradient

For $A$ positive semidefinite:

$$A\boldsymbol{x} = \boldsymbol{b} \iff \underset{\boldsymbol{x}}{\text{minimize}}\ f(\boldsymbol{x}) := \frac{1}{2}\boldsymbol{x}^T A\boldsymbol{x} - \boldsymbol{b}^T\boldsymbol{x}$$

$$\nabla f(\boldsymbol{x}) = A\boldsymbol{x} - \boldsymbol{b} := \boldsymbol{r}(\boldsymbol{x})$$

# Conjugate Direction

Def.: A set of nonzero vectors $\{\boldsymbol{d}_0, \boldsymbol{d}_1, \ldots, \boldsymbol{d}_\ell\}$ is said to be **conjugate** with respect to the symmetric positive definite matrix $A$ if

$$\boldsymbol{d}_i^T A \boldsymbol{d}_j = 0, \qquad \text{for all } i \neq j$$

(the vectors are linearly independent. Generally, not hortogonal.)

Theorem: Given an arbitrary $\boldsymbol{x}_0 \in \mathbb{R}^n$ and a set of conjugate vectors $\{\boldsymbol{d}_0, \boldsymbol{d}_1, \ldots, \boldsymbol{d}_{n-1}\}$ the sequence $\{\boldsymbol{x}_k\}$ generated by

$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k$$

where $\alpha_k$ is the analytical solution of $\min_\alpha f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$ given by:

$$\alpha_k = -\frac{\boldsymbol{r}_k^T \boldsymbol{d}_k}{\boldsymbol{d}_k^T A \boldsymbol{d}_k}$$

(aka, conjugate direction algorithm) converges to the solution $\boldsymbol{x}^*$ of the linear system and minimization problem in at most $n$ steps.

Proof:

$$\min_{\alpha} f(\boldsymbol{x}_k + \alpha \boldsymbol{d}_k)$$

We can compute the derivative with respect to $\alpha$:

$$\begin{aligned}\frac{\partial}{\partial \alpha} f(\boldsymbol{x} + \alpha \boldsymbol{d}) &= \frac{\partial}{\partial \alpha}(\boldsymbol{x} + \alpha \boldsymbol{d})^T A(\boldsymbol{x} + \alpha \boldsymbol{d}) - \boldsymbol{b}^T(\boldsymbol{x} + \alpha \boldsymbol{d})(+c) \\ &= \boldsymbol{d}^T A(\boldsymbol{x} + \alpha \boldsymbol{d}) - \boldsymbol{d}^T \boldsymbol{b} \\ &= \boldsymbol{d}^T(A\boldsymbol{x} - \boldsymbol{b}) + \alpha \boldsymbol{d}^T A \boldsymbol{d}\end{aligned}$$

Setting $\frac{\partial f(\boldsymbol{x}+\alpha \boldsymbol{d})}{\partial \alpha} = 0$ results in:

$$\alpha_k = -\frac{\boldsymbol{d}_k^T(A\boldsymbol{x}_k - \boldsymbol{b})}{\boldsymbol{d}_k^T A \boldsymbol{d}_k} = -\frac{\boldsymbol{d}_k^T \boldsymbol{r}(\boldsymbol{x}_k)}{\boldsymbol{d}_k^T A \boldsymbol{d}_k} \tag{1}$$

Since the directions $\{\boldsymbol{d}_k\}$ are linearly independent, they must span the whole space $\mathbb{R}^n$. Hence, there is a set of scalars $\sigma_k$ such that:

$$\boldsymbol{x}^* - \boldsymbol{x}_0 = \sigma_0 \boldsymbol{d}_0 + \sigma_1 \boldsymbol{d}_1 + \ldots + \sigma_{n-1} \boldsymbol{d}_{n-1}$$

By premultiplying this expression by $\boldsymbol{d}_k^T A$ and using the conjugacy property, we obtain:

$$\sigma_k = \frac{\boldsymbol{d}_k^T A(\boldsymbol{x}^* - \boldsymbol{x}_0)}{\boldsymbol{d}_k^T A \boldsymbol{d}_k} \tag{2}$$

If $\boldsymbol{x}_k$ is generated by conjugate direction algorithm, then we have

$$\boldsymbol{x}_k = \boldsymbol{x}_0 + \alpha_0 \boldsymbol{d}_0 + \alpha_1 \boldsymbol{d}_1 + \ldots + \alpha_k \boldsymbol{d}_{k-1}$$

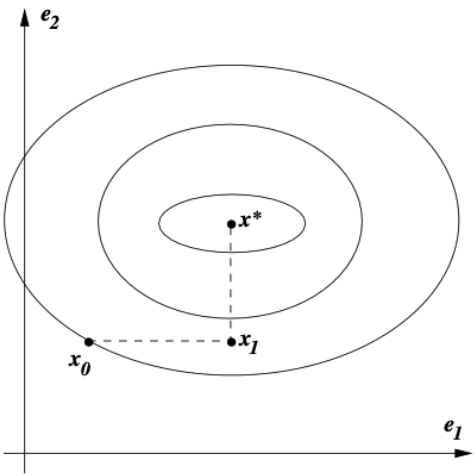
By premultiplying this expression by $\boldsymbol{d}_k^T A$ and using the conjugacy property, we have that

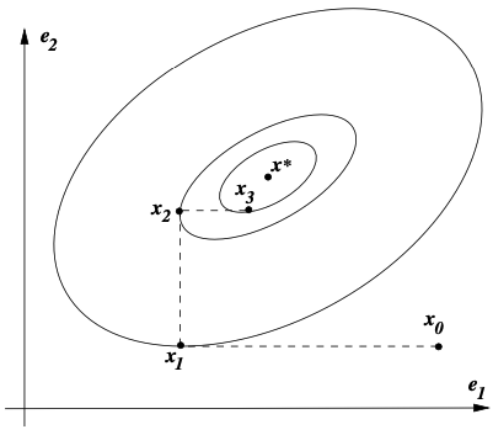
$$\boldsymbol{d}_k^T A(\boldsymbol{x}_k - \boldsymbol{x}_0) = 0$$

and therefore

$$\begin{aligned} \boldsymbol{d}_k^T A(\boldsymbol{x}^* - \boldsymbol{x}_0) = \boldsymbol{d}_k^T A(\boldsymbol{x}^* - \boldsymbol{x}_k + \boldsymbol{x}_k - \boldsymbol{x}_0) = \boldsymbol{d}_k^T A(\boldsymbol{x}^* - \boldsymbol{x}_k) + \boldsymbol{d}_k^T A(\boldsymbol{x}_k - \boldsymbol{x}_0) = \boldsymbol{d}_k^T A(\boldsymbol{x}^* - \boldsymbol{x}_k) \\ = \boldsymbol{d}_k^T (\boldsymbol{b} - A\boldsymbol{x}_k) = -\boldsymbol{d}_k^T \boldsymbol{r}_k. \end{aligned}$$

Using this result in (2) and comparing with (1) we conclude $\alpha_k = \sigma_k$. □


$e_2$
$x^*$
$x_1$
$x_0$
$e_1$


$e_2$
$x^*$
$x_3$
$x_2$
$x_0$
$x_1$
$e_1$

# Conjugate Gradient Method

The **conjugate gradient method** is a **conjugate direction method** with the property: In generating its set of conjugate vectors, it can compute a new vector $\boldsymbol{d}_k$ by using only the previous vector $\boldsymbol{d}_{k-1}$. Hence, fast.
remarkable property implies that the method requires little storage and computation.

$$\boldsymbol{d}_k = -\boldsymbol{r}_k + \beta_k \boldsymbol{d}_{k-1}$$

where $\beta_k$ is to be determined such that $\boldsymbol{d}_{k-1}$ and $\boldsymbol{d}_k$ must be conjugate with respect to $A$. By premultiplying by $\boldsymbol{d}_k^T A$ and imposing that $\boldsymbol{d}_{k-1}^T A \boldsymbol{d}_k = 0$ we find that

$$\beta_k = \frac{\boldsymbol{r}_k^T A \boldsymbol{d}_{k-1}}{\boldsymbol{d}_{k-1}^T A \boldsymbol{d}_{k-1}}$$

Larger values of $\beta$ indicate that the previous descent direction contributes more strongly.

# Algorithm CG

**Input:** $f, \boldsymbol{x}_0$
**Output:** $x^*$
Set $\boldsymbol{r}_0 \leftarrow A\boldsymbol{x}_0 - b, \boldsymbol{d}_0 \leftarrow \boldsymbol{r}_0, k \leftarrow 0$;
**while** $\boldsymbol{r}_k \neq 0$ **do**
$\alpha_k \leftarrow -\frac{\boldsymbol{d}_k^T \boldsymbol{r}(\boldsymbol{x}_k)}{\boldsymbol{d}_k^T A \boldsymbol{d}_k}$;
$\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k$;
$\boldsymbol{r}_{k+1} \leftarrow A\boldsymbol{x}_{k+1} - \boldsymbol{b}$;
$\beta_{k+1} \leftarrow \frac{\boldsymbol{r}_{k+1}^T A \boldsymbol{d}_k}{\boldsymbol{d}_k^T A \boldsymbol{d}_k}$;
$\boldsymbol{d}_{k+1} \leftarrow -\boldsymbol{r}_{k+1} + \beta_{k+1} \boldsymbol{d}_k$;
$k \leftarrow k + 1$;

**Input:** $f, \boldsymbol{x}_0$
**Output:** $x^*$
Set $\boldsymbol{r}_0 \leftarrow A\boldsymbol{x}_0 - b, \boldsymbol{d}_0 \leftarrow \boldsymbol{r}_0, k \leftarrow 0$;
**while** $\boldsymbol{r}_k \neq 0$ **do**
$\alpha_k \leftarrow -\frac{\boldsymbol{r}(\boldsymbol{x}_k)^T \boldsymbol{r}(\boldsymbol{x}_k)}{\boldsymbol{d}_k^T A \boldsymbol{d}_k}$;
$\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k$;
$\boldsymbol{r}_{k+1} \leftarrow \boldsymbol{r}_k + \alpha_k A \boldsymbol{d}_k$;
$\beta_{k+1} \leftarrow \frac{\boldsymbol{r}_{k+1}^T \boldsymbol{r}_{k+1}}{\boldsymbol{r}_k^T \boldsymbol{r}_k}$;
$\boldsymbol{d}_{k+1} \leftarrow -\boldsymbol{r}_{k+1} + \beta_{k+1} \boldsymbol{d}_k$;
$k \leftarrow k + 1$;

- we never need to know the vectors $\boldsymbol{x}, \boldsymbol{r}$, and $\boldsymbol{d}$ for more than the last two iterations.
- major computational tasks: the matrix–vector product $A\boldsymbol{d}_k$, inner products $\boldsymbol{d}_k^T A \boldsymbol{d}_k$ and $\boldsymbol{r}_{k+1}^T \boldsymbol{r}_{k+1}$, and three vector sums

# NonLinear Conjugate Gradient Methods

- The conjugate gradient method can be applied to nonquadratic functions as well.
- Smooth, continuous functions behave like quadratic functions close to a local minimum
- but! we do not know the value of $A$ that best approximates $f$ around $\boldsymbol{x}_k$. Instead, several choices for $\beta_k$ tend to work well:
- Two changes:
  - $alp[ha_k$ is computed by solving a apporximate line search
  - the residual $\boldsymbol{r}$, (it was simply the gradient of $f$), must be replaced by the gradient of the nonlinear objective $f$.

# NonLinear Conjugate Gradient Methods

Fletcher-Reeves Method

**Input:** $f, \boldsymbol{x}_0$
**Output:** $x^*$
Evaluate $f_0 = f(\boldsymbol{x}_0), \nabla f_0 = \nabla f(\boldsymbol{x}_0)$;
Set $\boldsymbol{d}_0 \leftarrow -\nabla f_0, k \leftarrow 0$;
**while** $\nabla f_k \neq 0$ **do**
  Compute $\alpha_k$ by line search and set
  $\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k$;
  Evaluate $\nabla f_{k+1}$;
  $\beta_{k+1}^{FR} \leftarrow \frac{\nabla f_{k+1}^T \nabla f_{k+1}}{\nabla f_k^T \nabla f_k}$;
  $\boldsymbol{d}_{k+1} \leftarrow -\nabla f_{k+1} + \beta_{k+1}^{FR} \boldsymbol{d}_k$;
  $k \leftarrow k + 1$;

Polak-Ribi'ere

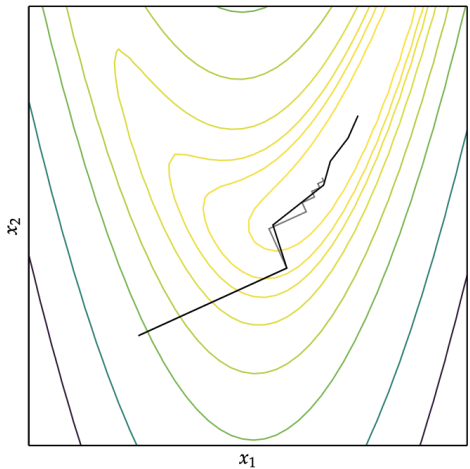
**Input:** $f, \boldsymbol{x}_0$
**Output:** $x^*$
Evaluate $f_0 = f(\boldsymbol{x}_0), \nabla f_0 = \nabla f(\boldsymbol{x}_0)$;
Set $\boldsymbol{d}_0 \leftarrow -\nabla f_0, k \leftarrow 0$;
**while** $\nabla f_k \neq 0$ **do**
  Compute $\alpha_k$ by line search and set
  $\boldsymbol{x}_{k+1} \leftarrow \boldsymbol{x}_k + \alpha_k \boldsymbol{d}_k$;
  Evaluate $\nabla f_{k+1}$;
  $\beta_{k+1}^{PR} \leftarrow \frac{\nabla f_{k+1}^T (\nabla f_{k+1} - \nabla f_k)}{\nabla f_k^T \nabla f_k}$;
  $\boldsymbol{d}_{k+1} \leftarrow -\nabla f_{k+1} + \beta_{k+1}^{FR} \boldsymbol{d}_k$;
  $k \leftarrow k + 1$;

PR with:

$$\beta_{k+1}^{+} = \max\{\beta_{k+1}^{PR}, 0\}$$

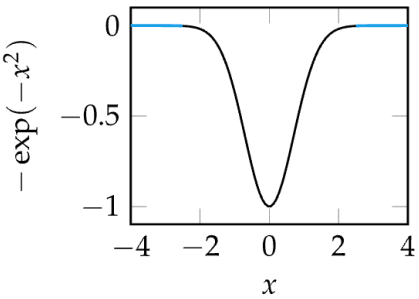
becomes $PR^+$ and guaranteed to converge (satisfy first Wolfe conditions.

The conjugate gradient method with the Polak-Ribière up- date. Gradient descent is shown in gray.



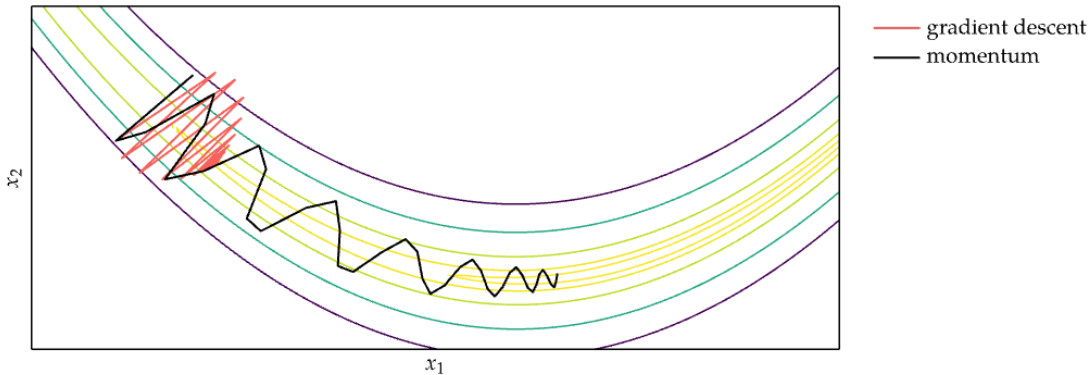
# Momentum and Accelerated Descent

- Addresses common convergence issues
- Some functions have regions with very small gradients (flat surface) where gradient descent gets stuck



# Momentum and Accelerated Descent

Rosenbrock function with $b = 100$




Momentum overcomes these issues by replicating the effect of physical momentum

# Momentum and Accelerated Descent

Momentum update equations:

$$\boldsymbol{v}_{k+1} = \beta \boldsymbol{v}_k - \alpha \nabla f(\boldsymbol{x}_k)$$
$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{v}_{k+1}$$

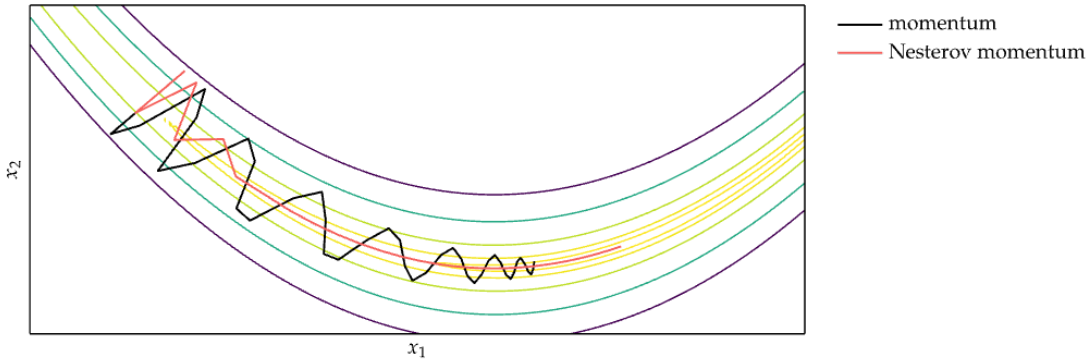
```
import numpy as np

class Momentum(DescentMethod):
  alpha: float # learning rate
  beta: float # momentum decay
  v: np.array # momentum

  def __init__(self, alpha, beta, f, grad, x):
    self.alpha = alpha
    self.beta = beta
    self.v = np.zeros_like(x)

  def step(self, grad, x):
    self.v = self.beta * self.v - self.alpha * grad(x)
    return x + self.v
```

# Nesterov Momentum and Accelerated Descent

Issue of momentum: steps do not slow down enough at the bottom of a valley, overshoot.

**Nesterov Momentum** update equations:

$$\boldsymbol{v}_{k+1} = \beta \boldsymbol{v}_k - \alpha \nabla f(\boldsymbol{x}_k + \beta \boldsymbol{v}_k)$$
$$\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{v}_{k+1}$$



# Adagrad

- Instead of using the same learning rate for all components of $\boldsymbol{x}$, **Adaptive Subgradient method** (Adagrad) adapts the learning rate for each component of $\boldsymbol{x}$. For each component of $\boldsymbol{x}$, the update equation is

$$x_{i,k+1} = x_{i,k} - \frac{\alpha}{\epsilon + \sqrt{s_{i,k}}} \nabla f_i(\boldsymbol{x}_k)$$

where

$$s_{i,k} = \sum_{j=1}^{k} (\nabla f_i(\boldsymbol{x}_j))^2$$

$$\epsilon \approx 1 \times 10^{-8}, \alpha = 0.01$$

- components of $\boldsymbol{s}$ are strictly nondecreasing, hence learning rate decreases over time

# RMSProp

- Extends Adagrad to avoid monotonically decreasing learning rate by maintaining a decaying average of squared gradients

$$\hat{s}_{k+1} = \gamma \hat{s}_k + (1-\gamma)\left(\nabla f_(\boldsymbol{x}_k) \odot \nabla f(\boldsymbol{x}_k)\right), \qquad \gamma \in [0,1], \qquad \odot \text{ element-wise product}$$

Update Equation

$$\begin{aligned} x_{i,k+1} &= x_{i,k} - \frac{\alpha}{\epsilon + \sqrt{\hat{s}_{i,k}}} \nabla f_i(\boldsymbol{x}_k) \\ &= x_{i,k} - \frac{\alpha}{\epsilon + RMS(\nabla f_i(\boldsymbol{x}_k))} \nabla f_i(\boldsymbol{x}_k) \end{aligned}$$

root mean square: For $n$ values $\{x_1, x_2, \ldots, x_n\}$

$$x_{\text{RMS}} = \sqrt{\frac{1}{n}\left({x_1}^2 + {x_2}^2 + \cdots + {x_n}^2\right)}.$$

# AdaDelta

Also extends Adagrad to avoid monotonically decreasing learning rate
Modifies RMSProp to eliminate learning rate parameter entirely

$$x_{i,k+1} = x_{i,k} - \frac{RMS(\Delta x_i)}{\epsilon + RMS(\nabla f_i(\boldsymbol{x}))} \nabla f_i(\boldsymbol{x}_k)$$

# Adam

- The **adaptive moment estimation method** (Adam), adapts the learning rate to each parameter.
- stores both an exponentially decaying gradient like momentum and an exponentially decaying squared gradient like RMSProp and Adadelta
- At each iteration, a sequence of values are computed

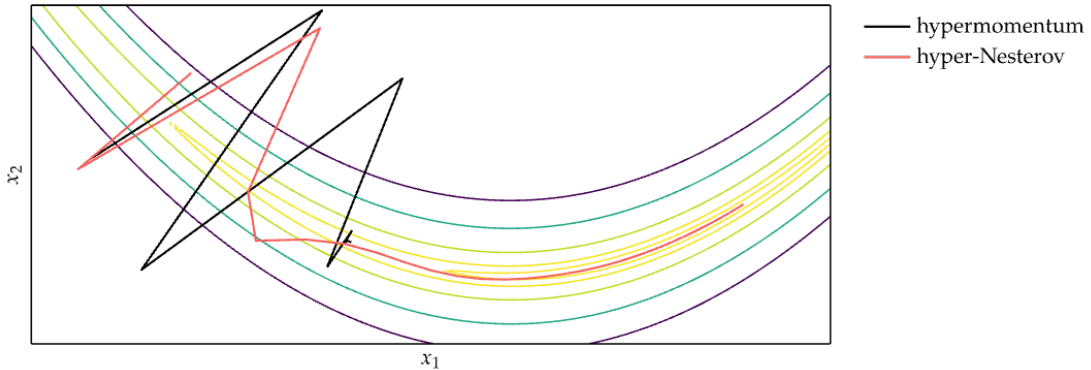
$$
\begin{aligned}
\text{Biased decaying momentum} \quad & \boldsymbol{v}_{k+1} = \beta \boldsymbol{v}_k - \alpha \nabla f(\boldsymbol{x}_k) \\
\text{Biased decaying squared gradient} \quad & \boldsymbol{s}_{k+1} = \gamma \boldsymbol{s}_k + (1-\gamma)\left(\nabla f_(\boldsymbol{x}_k) \odot \nabla f(\boldsymbol{x}_k)\right) \\
\text{Corrected decaying momentum} \quad & \hat{v}_{k+1} = \boldsymbol{v}_{k+1}/(1-\gamma_{v,k}) \\
\text{Corrected decaying squared gradient} \quad & \hat{s}_{k+1} = \boldsymbol{s}_{k+1}/(1-\gamma_{s,k}) \\
\text{Next iterate} \quad & \boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha \hat{v}_{k+1}/(\epsilon + \sqrt{\hat{s}_{k+1}})
\end{aligned}
$$

- Defaults: $\alpha = 0.001, \gamma_v = 0.9, \gamma_s = 0.999, \epsilon = 1 \times 10^{-8}$

# Hypergradient Descent

- Learning rate determines how sensitive the method is to the gradient signal.
- Many accelerated descent methods are highly sensitive to hyperparameters such as learning rate.
- Applying gradient descent to a hyperparameter of an underlying descent method is called hypergradient descent
- Requires computing the partial derivative of the objective function with respect to the hyperparameter

# Hypergradient Descent



# Summary

- Gradient descent follows the direction of steepest descent.
- The conjugate gradient method can automatically adjust to local valleys.
- Descent methods with momentum build up progress in favorable directions.
- A wide variety of accelerated descent methods use special techniques to speed up descent.
- Hypergradient descent applies gradient descent to the learning rate of an underlying descent method.