

AI505  
Optimization

## First-Order Methods

Marco Chiarandini

Department of Mathematics & Computer Science  
University of Southern Denmark

# Outline

Gradient Descent  
Conjugate Descent  
Accelerated Descents

1. Gradient Descent
2. Conjugate Descent
3. Accelerated Descents

# Descent Direction Methods

How to select the descent direction?

- first-order methods that rely on gradient
- second-order methods that rely on Hessian information

Advantages of first order methods:

- cheap iterations: good for small and large scale optimization embedded optimization
- helpful because easy to warm restart

Limitations of first order methods:

- not hard to find challenging instances for them.
- can converge slowly.

# Outline

Gradient Descent  
Conjugate Descent  
Accelerated Descents

1. Gradient Descent

2. Conjugate Descent

3. Accelerated Descents

# Gradient Descent

The **steepest descent** direction at  $\mathbf{x}_k$ , at  $k$ th iteration of a **local descent iterative method**, is the one opposite to the gradient (**gradient descent**):

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

Guaranteed to lead to improvement if:

- $f$  is smooth
- step size is sufficiently small
- $\mathbf{x}_k$  is not a stationary point (ie,  $\nabla f(\mathbf{x}_k) \neq 0$ )

# Gradient Descent: Example

- Suppose we have

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

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

- $\mathbf{x}_k = [1, 2]$

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

# Implementation

```
class DescentMethod:
    alpha: float

class GradientDescent(DescentMethod):
    def __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

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

Proof:

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

$$\nabla f(\mathbf{x}_k + \alpha_k^* \mathbf{d}_k) = \nabla_{\mathbf{d}_k} f(\mathbf{x}_k) = 0$$

because  $\alpha_k^*$  is minimum

$$\nabla f(\mathbf{x}_k + \alpha_k^* \mathbf{d}_k)^T \mathbf{d}_k = 0$$

because directional derivative:  $\nabla_s f(\mathbf{x}) = \nabla f(\mathbf{x})^T \mathbf{s}$

$$\mathbf{d}_{k+1} = - \frac{\nabla f(\mathbf{x}_k + \alpha_k^* \mathbf{d}_k)}{\|\nabla f(\mathbf{x}_k + \alpha_k^* \mathbf{d}_k)\|}$$

gradient descent

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

$$\mathbf{d}_{k+1}^T \mathbf{d}_k = 0 \implies \mathbf{d}_{k+1} \perp \mathbf{d}_k$$



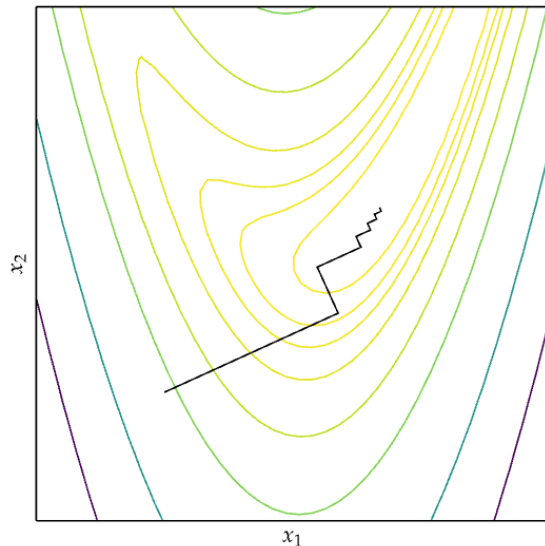


# Gradient Descent: Example

2D Rosenbrock function

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

Narrow valleys not aligned with gradient can be a problem



# Outline

Gradient Descent  
**Conjugate Descent**  
Accelerated Descents

1. Gradient Descent
2. Conjugate Descent
3. Accelerated Descents

# Conjugate Gradient

[Hestenes and Stiefel, 1950s]

For  $A$  symmetric positive definite:

$$Ax = b \iff \underset{x}{\text{minimize}} f(x) \stackrel{\text{def}}{=} \frac{1}{2}x^T Ax - b^T x$$

$$\nabla f(x) = Ax - b \stackrel{\text{def}}{=} r(x)$$

# Conjugate Direction

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

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

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

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

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

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

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

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

Proof:

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

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

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

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

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

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

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

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

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

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

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

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

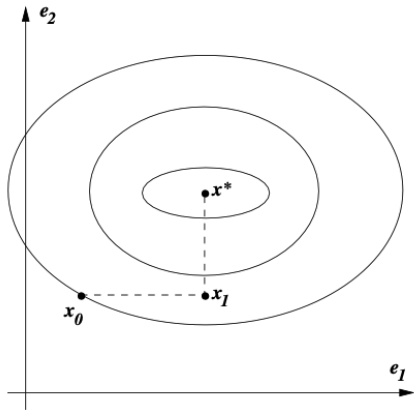
$$\mathbf{d}_k^T A(\mathbf{x}_k - \mathbf{x}_0) = 0$$

- and therefore

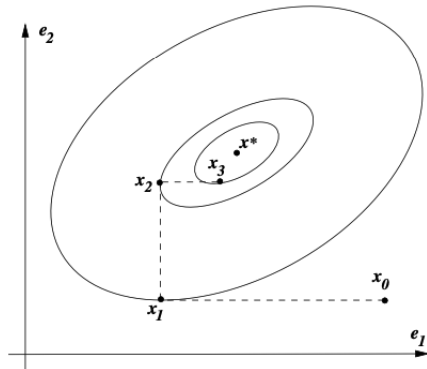
$$\begin{aligned} \mathbf{d}_k^T A(\mathbf{x}^* - \mathbf{x}_0) &= \mathbf{d}_k^T A(\mathbf{x}^* - \mathbf{x}_k + \mathbf{x}_k - \mathbf{x}_0) = \mathbf{d}_k^T A(\mathbf{x}^* - \mathbf{x}_k) + \mathbf{d}_k^T A(\mathbf{x}_k - \mathbf{x}_0) = \\ &= \mathbf{d}_k^T A(\mathbf{x}^* - \mathbf{x}_k) = \mathbf{d}_k^T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

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





If the matrix  $A$  is diagonal, the contours of the function  $f(\cdot)$  are ellipses whose axes are aligned with the coordinate directions



If  $A$  is not diagonal, its contours are elliptical, but they are usually not aligned with the coordinate directions.

Transform the problem to make  $A$  diagonal and minimize along the coordinate directions.

# 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  $\mathbf{d}_k$  by using only the previous vector  $\mathbf{d}_{k-1}$ . Hence, little storage and computation requirements.

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

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

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

- Larger values of  $\beta$  indicate that the previous descent direction contributes more strongly.
- Advantage with respect to steepest descent: implicitly reuses previous information about the function and thus better convergence.



# Algorithm CG

Basic version:

**Input:**  $f, \mathbf{x}_0$

**Output:**  $x^*$

Set  $\mathbf{r}_0 \leftarrow A\mathbf{x}_0 - \mathbf{b}$ ,  $\mathbf{d}_0 \leftarrow \mathbf{r}_0$ ,  $k \leftarrow 0$ ;

**while**  $\mathbf{r}_k \neq 0$  **do**

$$\alpha_k \leftarrow -\frac{\mathbf{d}_k^T \mathbf{r}_k}{\mathbf{d}_k^T A \mathbf{d}_k};$$

$$\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{d}_k;$$

$$\mathbf{r}_{k+1} \leftarrow A\mathbf{x}_{k+1} - \mathbf{b};$$

$$\beta_{k+1} \leftarrow \frac{\mathbf{r}_{k+1}^T A \mathbf{d}_k}{\mathbf{d}_k^T A \mathbf{d}_k};$$

$$\mathbf{d}_{k+1} \leftarrow -\mathbf{r}_{k+1} + \beta_{k+1} \mathbf{d}_k;$$

$$k \leftarrow k + 1;$$

Computationally improved version:

**Input:**  $f, \mathbf{x}_0$

**Output:**  $x^*$

Set  $\mathbf{r}_0 \leftarrow A\mathbf{x}_0 - \mathbf{b}$ ,  $\mathbf{d}_0 \leftarrow \mathbf{r}_0$ ,  $k \leftarrow 0$ ;

**while**  $\mathbf{r}_k \neq 0$  **do**

$$\alpha_k \leftarrow -\frac{\mathbf{r}_k^T \mathbf{r}_k}{\mathbf{d}_k^T A \mathbf{d}_k};$$

$$\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{d}_k;$$

$$\mathbf{r}_{k+1} \leftarrow \mathbf{r}_k + \alpha_k A \mathbf{d}_k;$$

$$\beta_{k+1} \leftarrow \frac{\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}}{\mathbf{r}_k^T \mathbf{r}_k};$$

$$\mathbf{d}_{k+1} \leftarrow -\mathbf{r}_{k+1} + \beta_{k+1} \mathbf{d}_k;$$

$$k \leftarrow k + 1;$$

- we never need to know the vectors  $\mathbf{x}$ ,  $\mathbf{r}$ , and  $\mathbf{d}$  for more than the last two iterations.
- major computational tasks: the matrix–vector product  $A \mathbf{d}_k$ , inner products  $\mathbf{d}_k^T A \mathbf{d}_k$  and  $\mathbf{r}_{k+1}^T \mathbf{r}_{k+1}$ , 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  $\mathbf{x}_k$ . Instead, several choices for  $\beta_k$  tend to work well:
- Two changes:
  - $\alpha_k$  is computed by solving an approximate line search
  - the residual  $\mathbf{r}$ , (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:**  $f, \mathbf{x}_0$

**Output:**  $\mathbf{x}^*$

Evaluate  $f_0 = f(\mathbf{x}_0), \nabla f_0 = \nabla f(\mathbf{x}_0)$ ;

Set  $\mathbf{d}_0 \leftarrow -\nabla f_0, k \leftarrow 0$ ;

**while**  $\nabla f_k \neq 0$  **do**

    Compute  $\alpha_k$  by line search and set

$\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{d}_k$ ;

    Evaluate  $\nabla f_{k+1}$ ;

$\beta_{k+1}^{FR} \leftarrow \frac{\nabla f_{k+1}^T \nabla f_{k+1}}{\nabla f_k^T \nabla f_k}$ ;

$\mathbf{d}_{k+1} \leftarrow -\nabla f_{k+1} + \beta_{k+1}^{FR} \mathbf{d}_k$ ;

$k \leftarrow k + 1$ ;

Polak-Ribière:

**Input:**  $f, \mathbf{x}_0$

**Output:**  $\mathbf{x}^*$

Evaluate  $f_0 = f(\mathbf{x}_0), \nabla f_0 = \nabla f(\mathbf{x}_0)$ ;

Set  $\mathbf{d}_0 \leftarrow -\nabla f_0, k \leftarrow 0$ ;

**while**  $\nabla f_k \neq 0$  **do**

    Compute  $\alpha_k$  by line search and set

$\mathbf{x}_{k+1} \leftarrow \mathbf{x}_k + \alpha_k \mathbf{d}_k$ ;

    Evaluate  $\nabla f_{k+1}$ ;

$\beta_{k+1}^{PR} \leftarrow \frac{\nabla f_{k+1}^T (\nabla f_{k+1} - \nabla f_k)}{\nabla f_k^T \nabla f_k}$ ;

$\mathbf{d}_{k+1} \leftarrow -\nabla f_{k+1} + \beta_{k+1}^{PR} \mathbf{d}_k$ ;

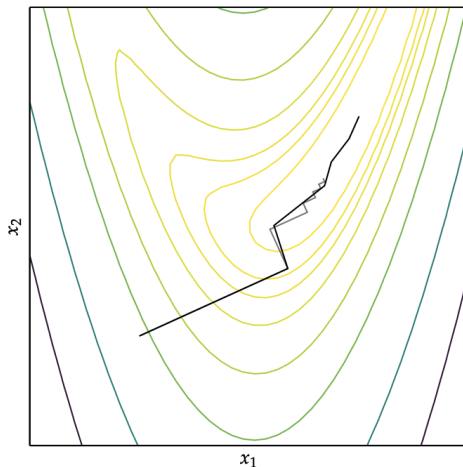
$k \leftarrow k + 1$ ;

PR with:

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

becomes PR<sup>+</sup> and guaranteed to converge (satisfies first Wolfe conditions).

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



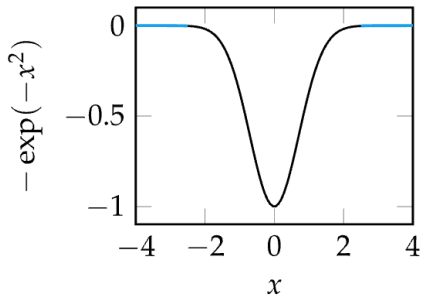
# Outline

Gradient Descent  
Conjugate Descent  
**Accelerated Descents**

1. Gradient Descent
2. Conjugate Descent
3. Accelerated Descents

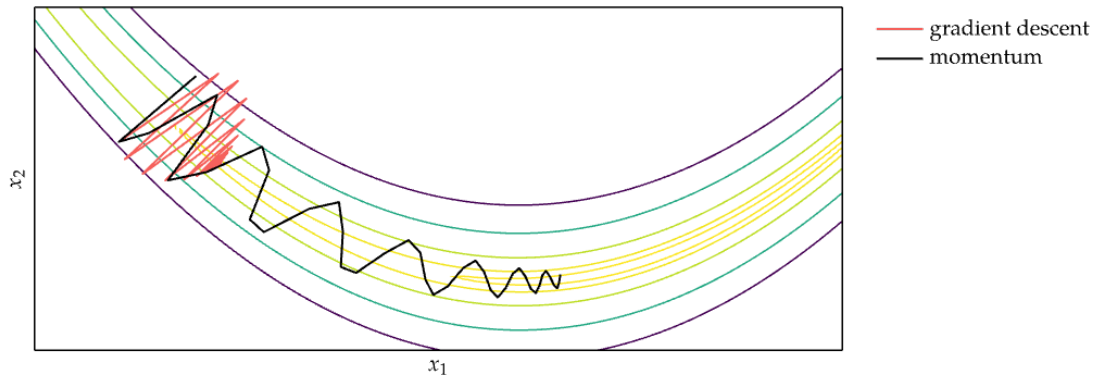
# Accelerated Descents

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



# Momentum

Rosenbrock function with  $b = 100$



Momentum overcomes these issues by replicating the effect of physical momentum

# Momentum

Momentum update equations:


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

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{v}_{k+1}$$

```
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
```



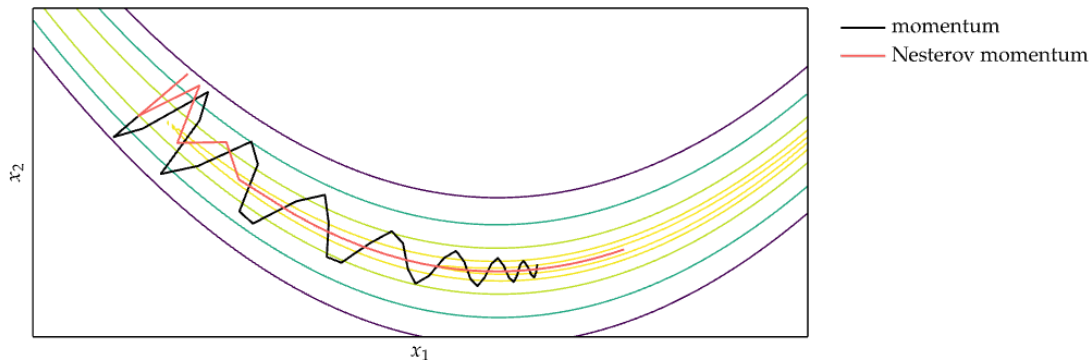
# Nesterov Momentum

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

**Nesterov Momentum** update equations:

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

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{v}_{k+1}$$



# Adagrad

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

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

where

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

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

- components of  $\mathbf{s}$  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

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

Update Equation

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

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

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

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

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

- 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

Biased decaying momentum	$\mathbf{v}_{k+1} = \beta \mathbf{v}_k - \alpha \nabla f(\mathbf{x}_k)$
Biased decaying squared gradient	$\mathbf{s}_{k+1} = \gamma \mathbf{s}_k + (1 - \gamma) (\nabla f(\mathbf{x}_k) \odot \nabla f(\mathbf{x}_k))$
Corrected decaying momentum	$\hat{\mathbf{v}}_{k+1} = \mathbf{v}_{k+1} / (1 - \gamma_v, k)$
Corrected decaying squared gradient	$\hat{\mathbf{s}}_{k+1} = \mathbf{s}_{k+1} / (1 - \gamma_s, k)$
Next iterate	$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha \hat{\mathbf{v}}_{k+1} / (\epsilon + \sqrt{\hat{\mathbf{s}}_{k+1}})$

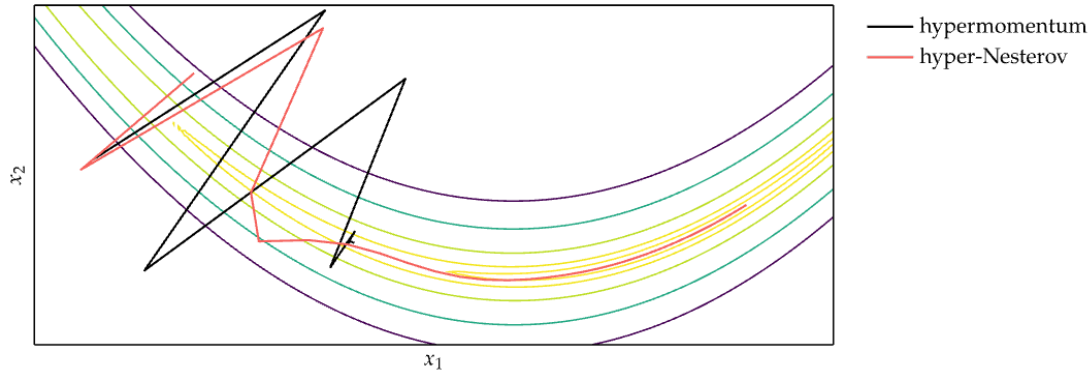
- Defaults:  $\alpha = 0.001, \gamma_v = 0.9, \gamma_s = 0.999, \epsilon = 1 \times 10^{-8}$

# 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

Gradient Descent  
Conjugate Descent  
Accelerated Descents



# 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.