EEE589
OPTIMIZATION
CH V – FIRST ORDER METHODS

Introduction

- In the previous chapter, the general concept of descent direction methods is introduced.
- This chapter discusses a variety of algorithms that use *first-order* methods to select the appropriate descent direction.
- First-order methods rely on gradient information to help direct the search for a minimum, which can be obtained using methods outlined in chapter 2

Gradient Descent

- An intuitive choice for descent direction **d** is the direction of steepest descent.
- Following the direction of steepest descent is guaranteed to lead to improvement, provided that the objective function is smooth, the step size is sufficiently small, and we are not already at a point where the gradient is zero.
- The direction of steepest descent is the direction opposite the gradient ∇f , hence the name *gradient descent*. For convenience in this chapter, we define

$$\mathbf{g}^{(k)} = \nabla f(\mathbf{x}^{(k)})$$

where $x^{(k)}$ is our design point at descent iteration k.

Gradient Descent

- In gradient descent, we typically normalize the direction of steepest descent

$$\mathbf{g}^{(k)} = \nabla f(\mathbf{x}^{(k)})$$

$$\mathbf{d}^{(k)} = -\frac{\mathbf{g}^{(k)}}{\|\mathbf{g}^{(k)}\|}$$

Suppose we have $f(\mathbf{x}) = x_1 x_2^2$. The gradient is $\nabla f = [x_2^2, 2x_1x_2]$. For $\mathbf{x}^{(k)} = [1, 2]$ we get an unnormalized direction of steepest descent $\mathbf{d} = [-4, -4]$, which is normalized to $\mathbf{d} = [-\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}]$.

Gradient Descent

- Jagged search paths result if we choose a step size that leads to the maximal decrease in f . In fact, the next direction will always be orthogonal to the current direction. We can show this as follows:

If we optimize the step size at each step, we have

$$\alpha^{(k)} = \arg\min_{\alpha} f(\mathbf{x}^{(k)} + \alpha \mathbf{d}^{(k)})$$

The optimization above implies that the directional derivative equals zero. Using equation (2.9), we have

$$\nabla f(\mathbf{x}^{(k)} + \alpha \mathbf{d}^{(k)})^\top \mathbf{d}^{(k)} = 0$$

We know

$$\mathbf{d}^{(k+1)} = -\frac{\nabla f(\mathbf{x}^{(k)} + \alpha \mathbf{d}^{(k)})}{\|\nabla f(\mathbf{x}^{(k)} + \alpha \mathbf{d}^{(k)})\|}$$

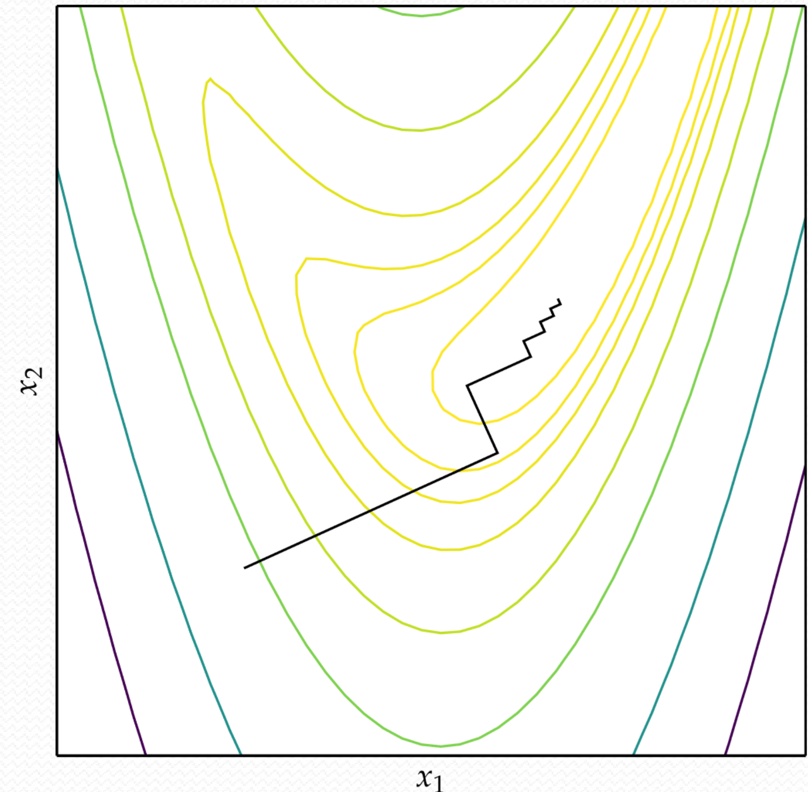
Hence,

$$\mathbf{d}^{(k+1)\top} \mathbf{d}^{(k)} = 0$$

which means that $\mathbf{d}^{(k+1)}$ and $\mathbf{d}^{(k)}$ are orthogonal.

Gradient Descent

- Narrow valleys aligned with a descent direction are not an issue.
- When the descent directions cross over the valley, many steps must be taken in order to make progress along the valley's floor as shown in figure.
- Gradient descent can result in zig-zagging in narrow canyons. Here we see the effect on the Rosenbrock function.
- An implementation of gradient descent is provided by algorithm.




```julia
abstract type DescentMethod end
struct GradientDescent <: DescentMethod
    α
end
init!(M::GradientDescent, f, ∇f, x) = M
function step!(M::GradientDescent, f, ∇f, x)
    α, g = M.α, ∇f(x)
    return x - α*g
end
```

Conjugate Gradient

- Gradient descent can perform poorly in narrow valleys. The *conjugate gradient* method overcomes this issue by borrowing inspiration from methods for optimizing quadratic functions:

$$\underset{\mathbf{x}}{\text{minimize}}\, f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b}^\top \mathbf{x} + c$$

where **A** is symmetric and positive definite, and thus f has a unique local minimum.

- The conjugate gradient method can optimize n-dimensional quadratic functions in n steps as shown in next figure. Its directions are *mutually conjugate* with respect to **A**:

$$\mathbf{d}^{(i)\top}\mathbf{A}\,\mathbf{d}^{(j)} = 0 \quad \text{for all } i \neq j$$

- The mutually conjugate vectors are the basis vectors of **A**. They are generally not orthogonal to one another.

Conjugate Gradient

- The successive conjugate directions are computed using gradient information and the previous descent direction. The algorithm starts with the direction of steepest descent:

$$\mathbf{d}^{(1)} = -\mathbf{g}^{(1)}$$

- We then use line search to find the next design point. For quadratic functions, the step factor α can be computed exactly. The update is then:

$$\mathbf{x}^{(2)} = \mathbf{x}^{(1)} + \alpha^{(1)}\mathbf{d}^{(1)}$$

Conjugate Gradient

$$\underset{\mathbf{x}}{\text{minimize}}\, f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{A}\mathbf{x} + \mathbf{b}^\top \mathbf{x} + c$$

Suppose we want to derive the optimal step factor for a line search on a quadratic function:

$$\underset{\alpha}{\text{minimize}}\, f(\mathbf{x} + \alpha \mathbf{d})$$

We can compute the derivative with respect to α:

$$\begin{aligned}
\frac{\partial f(\mathbf{x} + \alpha\mathbf{d})}{\partial \alpha} &= \frac{\partial}{\partial \alpha}\left[\frac{1}{2}(\mathbf{x} + \alpha\mathbf{d})^\top \mathbf{A}(\mathbf{x} + \alpha\mathbf{d}) + \mathbf{b}^\top(\mathbf{x} + \alpha\mathbf{d}) + c\right] \\
&= \mathbf{d}^\top \mathbf{A}(\mathbf{x} + \alpha\mathbf{d}) + \mathbf{d}^\top \mathbf{b} \\
&= \mathbf{d}^\top(\mathbf{A}\mathbf{x} + \mathbf{b}) + \alpha\mathbf{d}^\top \mathbf{A}\mathbf{d}
\end{aligned}$$

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

$$\alpha = -\frac{\mathbf{d}^\top(\mathbf{A}\mathbf{x} + \mathbf{b})}{\mathbf{d}^\top \mathbf{A}\mathbf{d}}$$

Conjugate Gradient

- Subsequent iterations choose $d^{(k+1)}$ based on the next gradient and a contribution from the current descent direction:

$$\mathbf{d}^{(k+1)} = -\mathbf{g}^{(k+1)} + \beta^{(k)}\mathbf{d}^{(k)}$$

for scalar parameter β. Larger values of β indicate that the previous descent direction contributes more strongly.

Conjugate Gradient

We can derive the best value for β for a known $\mathbf{A}$, using the fact that $\mathbf{d}^{(k+1)}$ is conjugate to $\mathbf{d}^{(k)}$:

$$\begin{aligned}
\mathbf{d}^{(k+1)\top}\mathbf{A}\mathbf{d}^{(k)} &= 0 \\
\Rightarrow (-\mathbf{g}^{(k+1)} + \beta^{(k)}\mathbf{d}^{(k)})^\top\mathbf{A}\mathbf{d}^{(k)} &= 0 \\
\Rightarrow -\mathbf{g}^{(k+1)\top}\mathbf{A}\mathbf{d}^{(k)} + \beta^{(k)}\mathbf{d}^{(k)\top}\mathbf{A}\mathbf{d}^{(k)} &= 0 \\
\Rightarrow \beta^{(k)} = \frac{\mathbf{g}^{(k+1)\top}\mathbf{A}\mathbf{d}^{(k)}}{\mathbf{d}^{(k)\top}\mathbf{A}\mathbf{d}^{(k)}}
\end{aligned}$$

The conjugate gradient method can be applied to nonquadratic functions as well. Smooth, continuous functions behave like quadratic functions close to a local minimum, and the conjugate gradient method will converge very quickly in such regions.

Conjugate Gradient

Unfortunately, we do not know the value of $\mathbf{A}$ that best approximates f around $\mathbf{x}^{(k)}$. Instead, several choices for $\beta^{(k)}$ tend to work well:

Fletcher-Reeves:[2]

$$\beta^{(k)} = \frac{\mathbf{g}^{(k)\top}\mathbf{g}^{(k)}}{\mathbf{g}^{(k-1)\top}\mathbf{g}^{(k-1)}}$$

Polak-Ribière:[3]

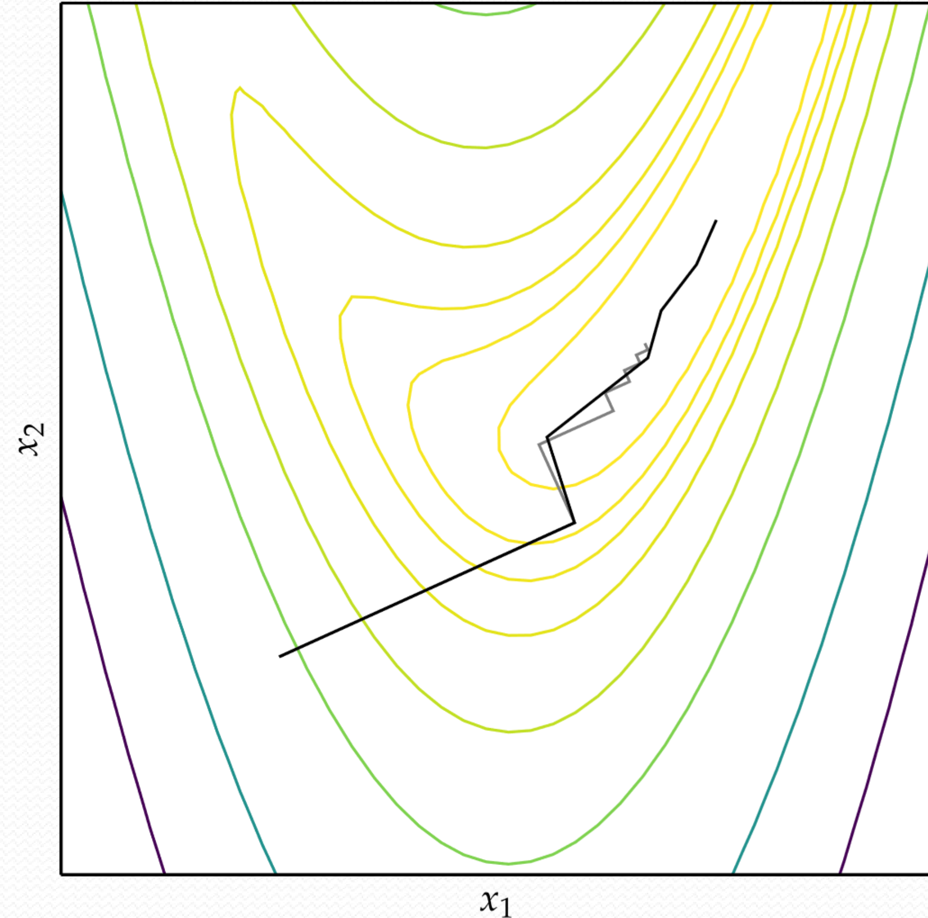
$$\beta^{(k)} = \frac{\mathbf{g}^{(k)\top}\left(\mathbf{g}^{(k)} - \mathbf{g}^{(k-1)}\right)}{\mathbf{g}^{(k-1)\top}\mathbf{g}^{(k-1)}}$$

Convergence for the Polak-Ribière method (algorithm 5.2) can be guaranteed if we modify it to allow for automatic resets:

$$\beta \leftarrow \max(\beta, 0)$$

Conjugate Gradient

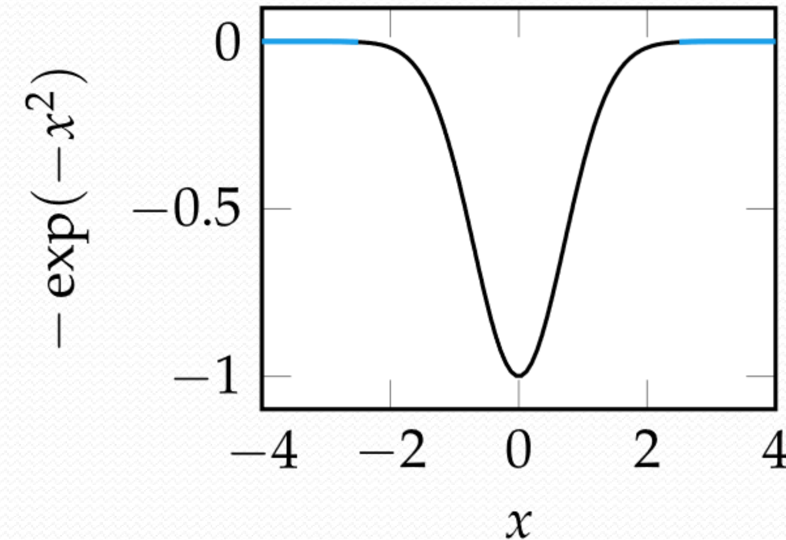
- Figure shows an example search using the conjugate gradient method with the Polak-Ribière update. Gradient descent is shown in gray.



Momentum

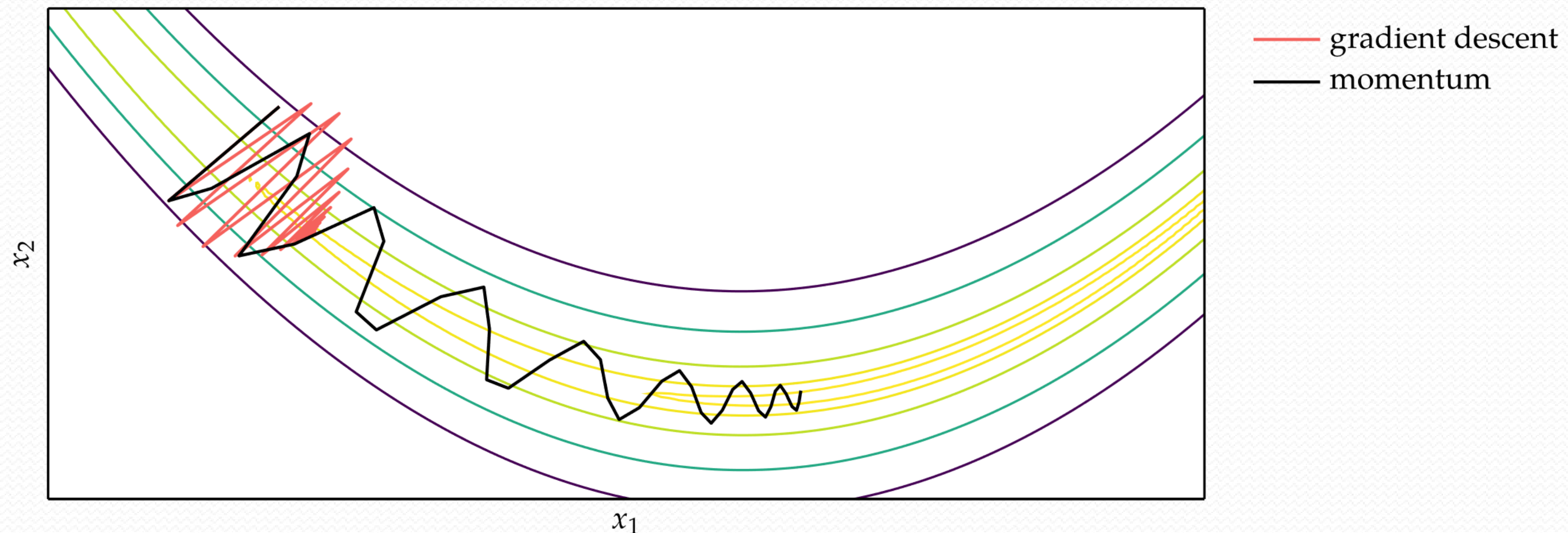
- Gradient descent will take a long time to traverse a nearly flat surface as shown in figure.
- Regions that are nearly flat have gradients with small magnitudes and can thus require many iterations of gradient descent to traverse.
- Allowing momentum to accumulate is one way to speed progress.
- We can modify gradient descent to incorporate momentum.
- The *momentum* update equations are:

$$\mathbf{v}^{(k+1)} = \beta \mathbf{v}^{(k)} - \alpha \mathbf{g}^{(k)}$$
$$\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{v}^{(k+1)}$$



Momentum

- For β = 0, we recover gradient descent.
- Momentum can be interpreted as a ball rolling down a nearly horizontal incline.
- The ball naturally gathers momentum as gravity causes it to accelerate, just as the gradient causes momentum to accumulate in this descent method.
- Momentum descent is compared to gradient descent in figure.



Nesterov Momentum

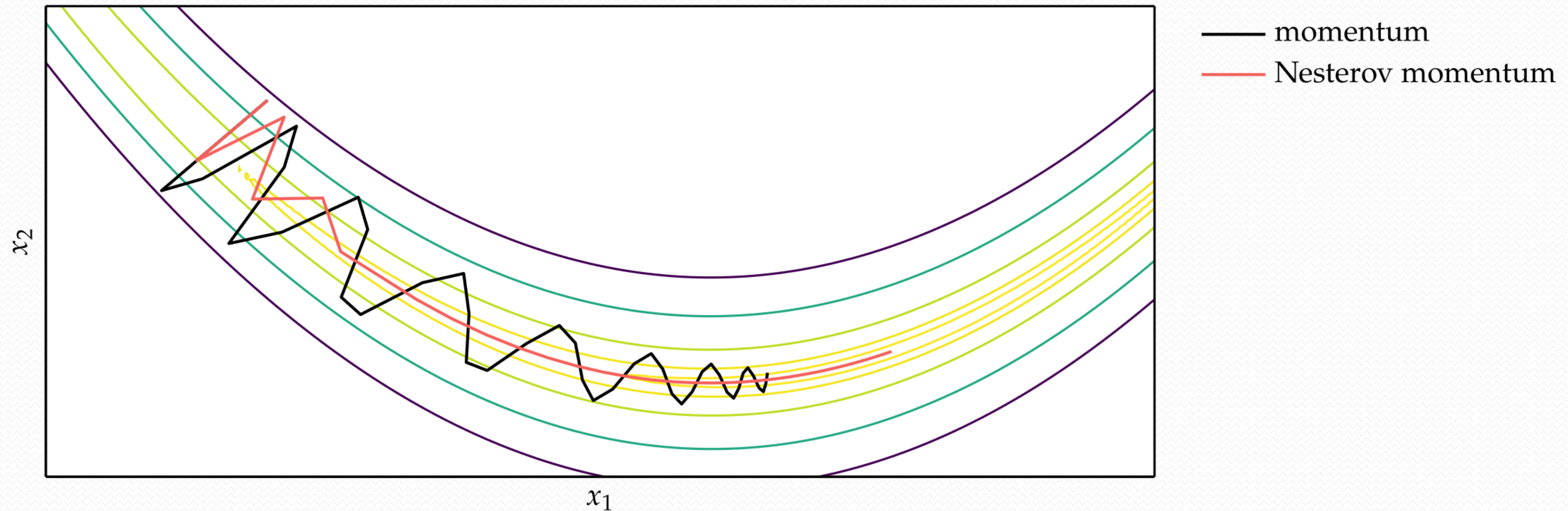
- One issue of momentum is that the steps do not slow down enough at the bottom of a valley and tend to overshoot the valley floor. *Nesterov momentum* modifies the momentum algorithm to use the gradient at the projected future position:

$$\mathbf{v}^{(\mathrm{k}+1)} = \beta \mathbf{v}^{(\mathrm{k})} - \alpha \nabla f\left(\mathbf{x}^{(\mathrm{k})} + \beta \mathbf{v}^{(\mathrm{k})}\right)$$

$$\mathbf{x}^{(\mathrm{k}+1)} = \mathbf{x}^{(\mathrm{k})} + \mathbf{v}^{(\mathrm{k}+1)}$$

Nesterov Momentum

- The Nesterov momentum and momentum descent methods are compared in figure.



Adagrad

- Momentum and Nesterov momentum update all components of $\mathbf{x}$ with the same learning rate.
- The *adaptive subgradient* method, or *Adagrad*, adapts a learning rate for each component of $\mathbf{x}$.
- Adagrad dulls the influence of parameters with consistently high gradients, thereby increasing the influence of parameters with infrequent updates.

Adagrad

- The Adagrad update step is:

$$x_i^{(k+1)} = x_i^{(k)} - \frac{\alpha}{\epsilon + \sqrt{s_i^{(k)}}} g_i^{(k)}$$

where

$$s_i^{(k)} = \sum_{j=1}^{k} \left(g_i^{(j)}\right)^2 \qquad \epsilon \approx 1 \times 10^{-8}$$

- Adagrad is far less sensitive to the learning rate parameter α.
- The learning rate parameter is typically set to a default value of 0.01.
- Adagrad's primary weakness is that the components of **s** are each strictly nondecreasing.
- The accumulated sum causes the effective learning rate to decrease during training, often becoming infinitesimally small before convergence.

RMSProp

- *RMSProp* extends Adagrad to avoid the effect of a monotonically decreasing learning rate. RMSProp maintains a decaying average of squared gradients. This average is updated according to:

$$\hat{\mathbf{s}}^{(k+1)} = \gamma \hat{\mathbf{s}}^{(k)} + (1-\gamma)\left(\mathbf{g}^{(k)} \odot \mathbf{g}^{(k)}\right)$$

where the decay $\gamma \in [0, 1]$ is typically close to 0.9.

- The decaying average of past squared gradients can be substituted into RMSProp's update equation:

$$\begin{aligned} \mathrm{x}_{\mathrm{i}}^{(\mathrm{k}+1)} &= x_{\mathrm{i}}^{(\mathrm{k})} - \frac{\alpha}{\epsilon + \sqrt{\hat{s}_i^{(k)}}} g_i^{(k)} \\ &= x_{\mathrm{i}}^{(\mathrm{k})} - \frac{\alpha}{\epsilon + \mathrm{RMS}(g_i)} g_i^{(k)} \end{aligned}$$

Adadelta

- *Adadelta* is another method for overcoming Adagrad's monotonically decreasing learning rate. After independently deriving the RMSProp update, the authors noticed that the units in the update equations for gradient descent, momentum, and Adagrad do not match. To fix this, they use an exponentially decaying average of the square updates:

$$x_i^{(k+1)} = x_i^{(k)} - \frac{\mathrm{RMS}(\Delta x_i)}{\epsilon + \mathrm{RMS}(g_i)} g_i^{(k)}$$

which eliminates the learning rate parameter entirely.

Adam

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

1. Biased decaying momentum
2. Biased decaying squared gradient
3. Corrected decaying momentum
4. Corrected decaying squared gradient
5. Next iterate $\mathbf{x}^{(k+1)}$

Hypergradient Descent

- The accelerated descent methods are either extremely sensitive to the learning rate or go to great lengths to adapt the learning rate during execution.
- The learning rate dictates how sensitive the method is to the gradient signal.
- A rate that is too high or too low often drastically affects performance.
- *Hypergradient descent* was developed with the understanding that the derivative of the learning rate should be useful for improving optimizer performance.
- A *hypergradient* is a derivative taken with respect to a hyperparameter.
- Hypergradient algorithms reduce the sensitivity to the hyperparameter, allowing it to adapt more quickly.

Hypergradient Descent

- Hypergradient descent applies gradient descent to the learning rate of an underlying descent method.
- The method requires the partial derivative of the objective function with respect to the learning rate.
- For gradient descent, this partial derivative is:

$$\begin{aligned}\frac{\partial f(\mathbf{x}^{(k)})}{\partial \alpha} &= (\mathbf{g}^{(k)})^\top \frac{\partial}{\partial \alpha}\left(\mathbf{x}^{(k-1)} - \alpha \mathbf{g}^{(k-1)}\right) \\ &= (\mathbf{g}^{(k)})^\top \left(-\mathbf{g}^{(k-1)}\right)\end{aligned}$$

Hypergradient Descent

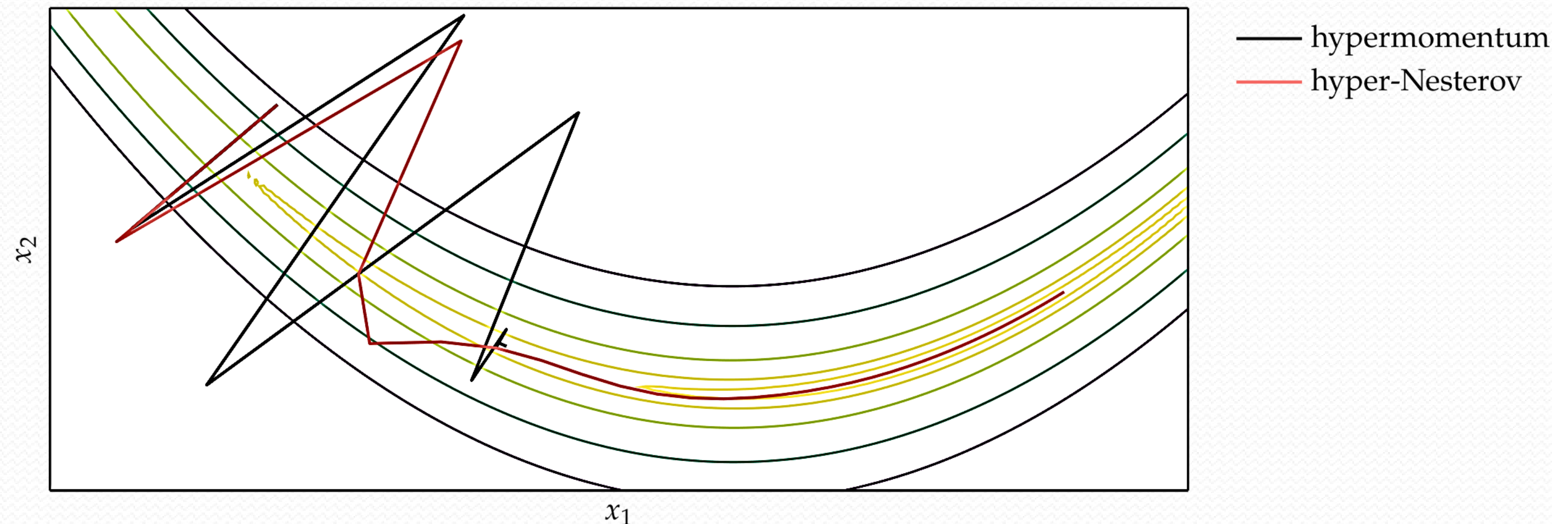
- Computing the hypergradient thus requires keeping track of the last gradient.
- The resulting update rule is:

$$\begin{aligned}\alpha^{(k+1)} &= \alpha^{(k)} - \mu \frac{\partial f(\mathbf{x}^{(k)})}{\partial \alpha} \\ &= \alpha^{(k)} + \mu (\mathbf{g}^{(k)})^\top \mathbf{g}^{(k-1)}\end{aligned}$$

where μ is the hypergradient learning rate.

Hypergradient Descent

- This derivation can be applied to any gradient-based descent method.
- Implementations are provided in the book for the hypergradient versions of gradient descent and Nesterov momentum.
- These methods are visualized in figure. The momentum and Nesterov momentum methods compared on the Rosenbrock function with b = 100.



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.