2-4 Gradient Descent Algorithms and Their Variations

Zhonglei Wang

WISE and SOE, XMU, 2025

Contents

Recall

1. Batch gradient descent

- Randomly initialize $\boldsymbol{\theta}^{(0)}$

- Based on the current parameter $\boldsymbol{\theta}^{(t)}$, obtain

$$\nabla \mathcal{J}\left(\boldsymbol{\theta}^{(t)}\right) = \frac{\partial \mathcal{J}}{\partial \boldsymbol{\theta}}(\boldsymbol{\theta}^{(t)}) = n^{-1} \sum_{i=1}^{n} \frac{\partial \mathcal{L}_i}{\partial \boldsymbol{\theta}}\left(\boldsymbol{\theta}^{(t)}\right)$$

  ▷ $\mathcal{L}_i$: loss function associated with the ith training example

- Update parameter

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \alpha \nabla \mathcal{J}\left(\boldsymbol{\theta}^{(t)}\right)$$

- Go back to Step 2 until convergence

Recall

1. Disadvantages
   - Computationally inefficient when n is large
2. Why not use part of the training examples for each iteration?

Mini-batch gradient descent

1. Idea: use part of training examples for each iteration
2. Partition the index set of training examples: $\{1, \ldots, n\} = S_1 \cup \cdots \cup S_k$
   - $|S_1| = \cdots = |S_{k-1}| = m$
   - $|S_k| \leq m$
   - $k = \lceil n/m \rceil$
   - m is usually a power of 2, in practice. For example, $m = 512$

Mini-batch gradient descent

1. For the tth iteration, update the model paramter by

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \alpha \nabla \mathcal{J}_t \left(\boldsymbol{\theta}^{(t)} \right)$$

$$\nabla \mathcal{J}_t \left(\boldsymbol{\theta}^{(t)} \right) = \left| S_{t\%k+1} \right|^{-1} \sum_{i \in S_{t\%k+1}} \frac{\partial \mathcal{L}_i}{\partial \boldsymbol{\theta}} (\boldsymbol{\theta}^{(t)})$$

   - Only use training examples in $S_{t\%k+1}$ to obtain the gradients

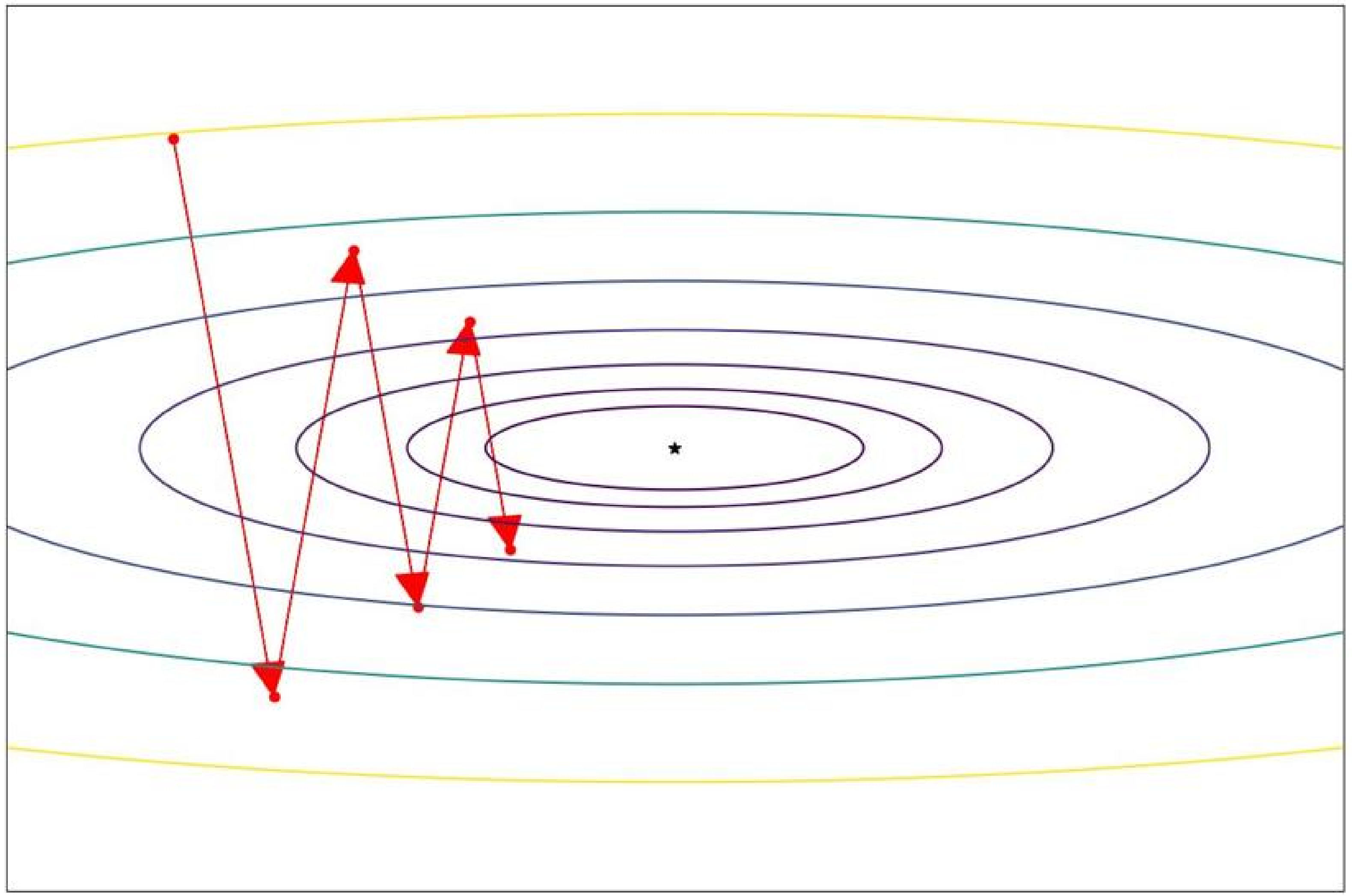
2. We finish one epoch when each training example is used

Discussion

1. Two special cases
   - $m = 1:$ stochastic gradient descent
   - $m = n:$ batch gradient descent
2. Mini-batch gradient descent sacrifices accuracy when $m < n$
3. Nevertheless, it saves memery and is computationally more efficient
4. Commonly used in practice
5. Next, we consider some efficient gradient descent algorithms.

Background

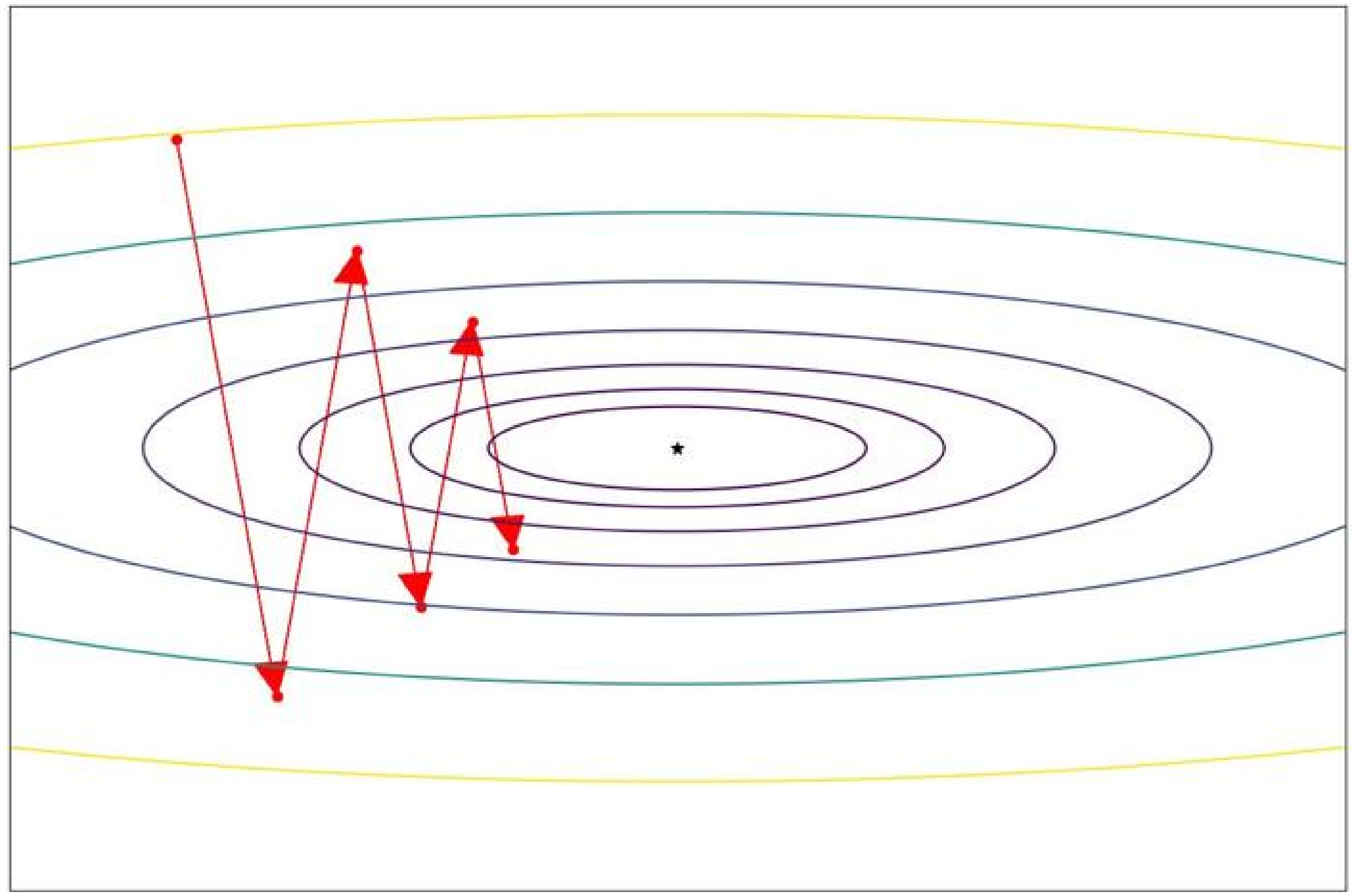
1. Gradient shows the fastest direction along which the cost function increases
2. It may causes problems, especially when the cost function is unstable



Background

1. For gradient descent algorithms, replace gradients by possibly more efficient vectors
2. To achieve this goal, we introduce
    - Momentum
    - RMSprop
    - Adam

Momentum



1. Ideally, we want to
   - Decrease the up-down effect
   - Increase the left-right effect

Momentum

1. Taking mean might be good?
   - Requires memory to store all the past gradients
   - Impossible for deep learning models
2. Consider EWMA (Exponential Weighted Moving Average)
   - Original sequence: $\{s_i : i = 1, 2, \ldots\}$
   - EWMA sequence: $\{v_i : i = 1, 2, \ldots\}$

$$v_i = \beta_1 v_{i-1} + (1 - \beta_1) s_i \quad (i = 1, 2, \ldots)$$

     - ▷ $v_0 = 0$
     - ▷ β_1 controls how much we stay with the "momentum" v_{i-1}

Computation

1. Let $\boldsymbol{g}^{(t)}$ be a gradient evaluated at the current parameter for the tth iteration
   - It can be $\mathrm{d}\boldsymbol{b}$ or $\mathrm{d}\boldsymbol{W}$ evaluated at the current parameter $\boldsymbol{\theta}^{(t)}$
   - We ignore the superscript for layer for simplicity
2. Obtain the EWMA "gradient" $\boldsymbol{v}_b^{(t)}$ as follows

$$\boldsymbol{v}_b^{(t)} = \beta_1 \boldsymbol{v}_b^{(t-1)} + (1 - \beta_1)\boldsymbol{g}^{(t)}$$

   - $\beta_1 = 0.9$: a hyperparameter, but seldom tuned
   - $\boldsymbol{v}_b^{(0)} = 0$: initial momentum

Computation

1. A fact

$$\frac{1}{(1-\beta_1)^{-1}}\sum_{i=1}^{\infty}\beta_1^i = 1$$

2. More details

$$\begin{aligned}\boldsymbol{v}^{(t)} &= \beta_1\boldsymbol{v}^{(t-1)} + (1-\beta_1)\boldsymbol{g}^{(t)}\\ &= (1-\beta_1)\beta_1^{t-1}\boldsymbol{g}^{(1)} + (1-\beta_1)\beta_1^{t-2}\boldsymbol{g}^{(2)} + \cdots + (1-\beta_1)\boldsymbol{g}^{(t)}\\ &= \frac{1}{(1-\beta_1)^{-1}}\sum_{i=1}^{t}\beta_1^{n-i}\boldsymbol{g}^{(i)}\end{aligned}$$

- When t is large, it is approximately weighted average
- $(1-\beta_1)^{-1}$: can be viewed as the "effective sample size" for EMWA

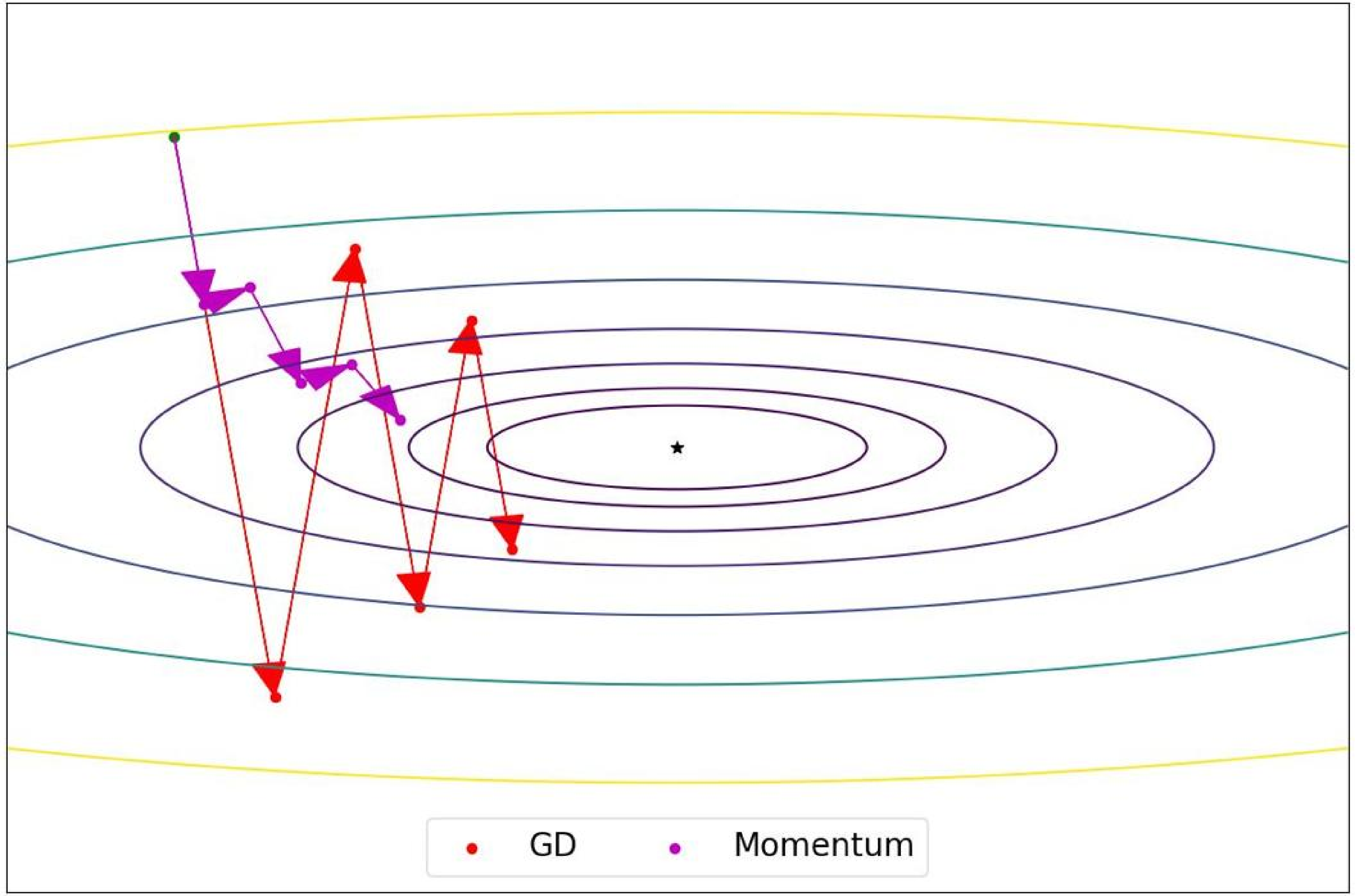
Momentum-based gradient descent algorithm

1. Randomly initialize $\boldsymbol{\theta}^{(0)}$
2. Based on the current model parameter $\boldsymbol{\theta}^{(t)}$, obtain $\mathrm{d}\boldsymbol{b}^{[1](t)}$
   - Take the update procedure for $\boldsymbol{b}^{[1]}$ as an example
   - The procedure applies to other parameters as well
3. Update

$$\boldsymbol{b}^{[1](t+1)} = \boldsymbol{b}^{[1](t)} - \alpha \boldsymbol{v}_b^{[1](t+1)}$$

   - $\boldsymbol{v}_b^{[1](t+1)} = \beta_1 \boldsymbol{v}_b^{[1](t)} + (1-\beta_1)\mathrm{d}\boldsymbol{b}^{[1](t)}$
   - $\boldsymbol{v}_b^{[1](0)} = 0$
   - $\beta_1 = 0.9$ by default
4. Go back to Step 2 until convergence

Comparison



Comparison

1. Momentum indeed decreases the variability in the up-down direction
2. Nevertheless, the convergence rate is slow
3. One possible solution: different learning rates for different directions
    - Low learning rate for the up-down direction
    - High learning rate for the left-right direction

RMSprop-based gradient descent algorithm

Step 1. Randomly initialize $\boldsymbol{\theta}^{(0)}$

Step 2. Based on the current model parameter $\boldsymbol{\theta}^{(t)}$, obtain $\mathrm{d}\boldsymbol{b}^{[1](t)}$

- Take the update procedure for $\boldsymbol{b}^{[1]}$ as an example

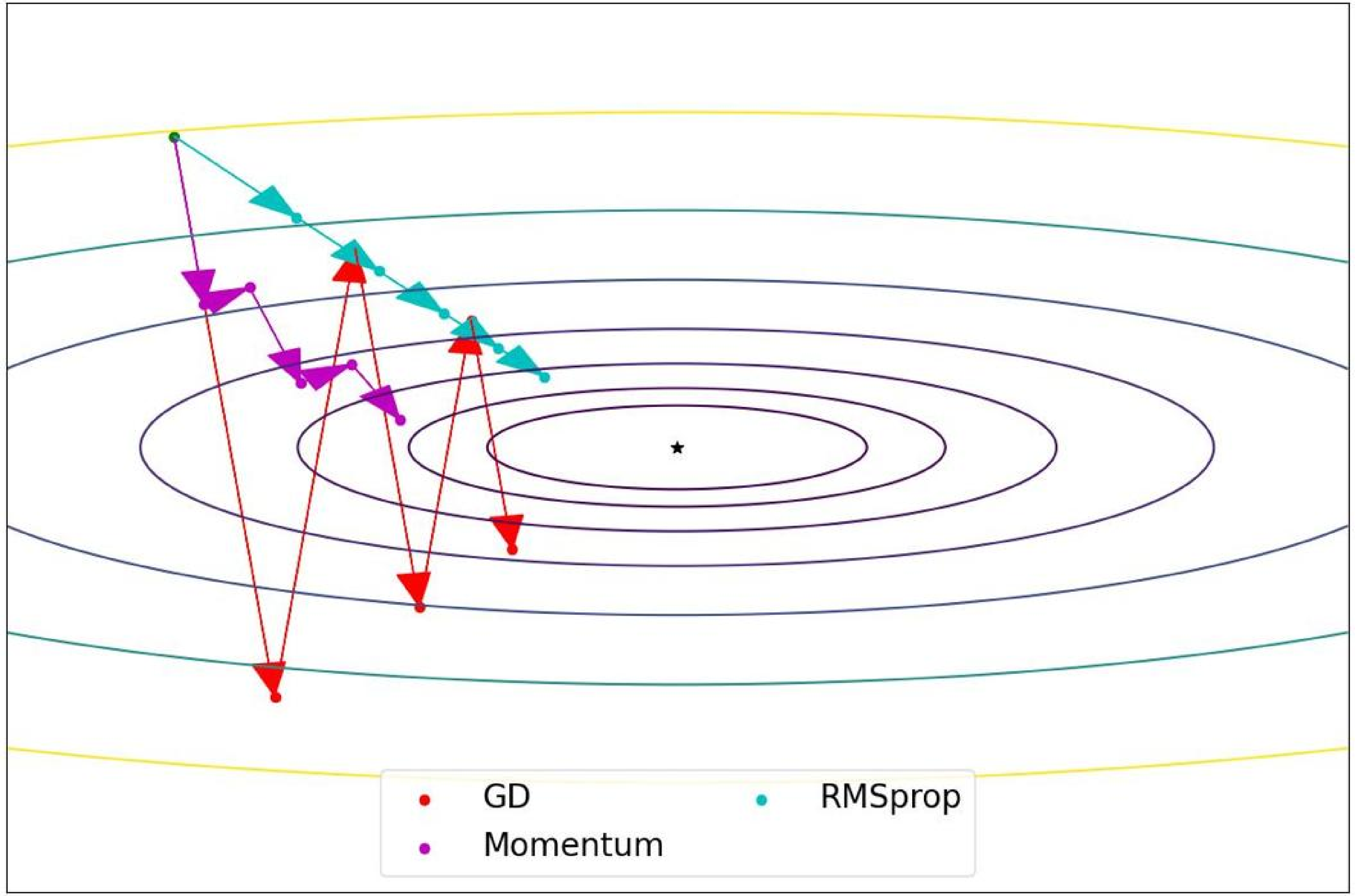
Step 3. Update

$$\boldsymbol{b}^{[1](t+1)} = \boldsymbol{b}^{[1](t)} - \frac{\alpha}{\sqrt{\epsilon + \boldsymbol{s}_b^{[1](t+1)}}} \mathrm{d}\boldsymbol{b}^{[1](t)}$$

- $\epsilon = 10^{-8}$ by default
- $\boldsymbol{s}_b^{[1](t+1)} = \beta_2 \boldsymbol{s}_b^{[1](t)} + (1 - \beta_2)\mathrm{d}\boldsymbol{b}^{[1](t)} \circ \mathrm{d}\boldsymbol{b}^{[1](t)}$
- $\boldsymbol{s}_b^{[1](0)} = 0$
- $\beta_2 = 0.99$ by default

Step 4. Go back to Step 2 until convergence

Comparison



Comparison

1. The path for RMSprop is more smooth
2. Nevertheless, the convergence rate is not that fast

Algorithms

1. Classical gradient descent algorithms

$$\boldsymbol{b}^{[1](t+1)} = \boldsymbol{b}^{[1](t)} - \alpha \mathrm{d}\boldsymbol{b}^{[1](t)}$$

2. Momentum only modifies the gradient part

$$\boldsymbol{b}^{[1](t+1)} = \boldsymbol{b}^{[1](t)} - \alpha \boldsymbol{v}_b^{[1](t+1)}$$

3. RMSprop only modifies the learning rate part

$$\boldsymbol{b}^{[1](t+1)} = \boldsymbol{b}^{[1](t)} - \frac{\alpha}{\sqrt{\epsilon + \boldsymbol{s}_b^{[1](t+1)}}} \mathrm{d}\boldsymbol{b}^{[1](t)}$$

4. Why not combine those two together?

Adam (Adaptive moment estimation)

1. Randomly initialize $\boldsymbol{\theta}^{(0)}$
2. Based on the current model parameter $\boldsymbol{\theta}^{(t)}$, obtain $\mathrm{d}\boldsymbol{b}^{[1](t)}$
   - Take the update procedure for $\boldsymbol{b}^{[1]}$ as an example
3. Update

$$\boldsymbol{b}^{[1](t+1)} = \boldsymbol{b}^{[1](t)} - \frac{\alpha}{\epsilon + \sqrt{\hat{\boldsymbol{s}}_b^{[1](t+1)}}} \hat{\boldsymbol{v}}_b^{[1](t+1)}$$

   - See next slide for $\hat{\boldsymbol{s}}_b^{[1](t+1)}$ and $\hat{\boldsymbol{v}}_b^{[1](t+1)}$
   - $\epsilon = 10^{-8}$ by default
4. Go back to Step 2 until convergence

Adam (Adaptive moment estimation)

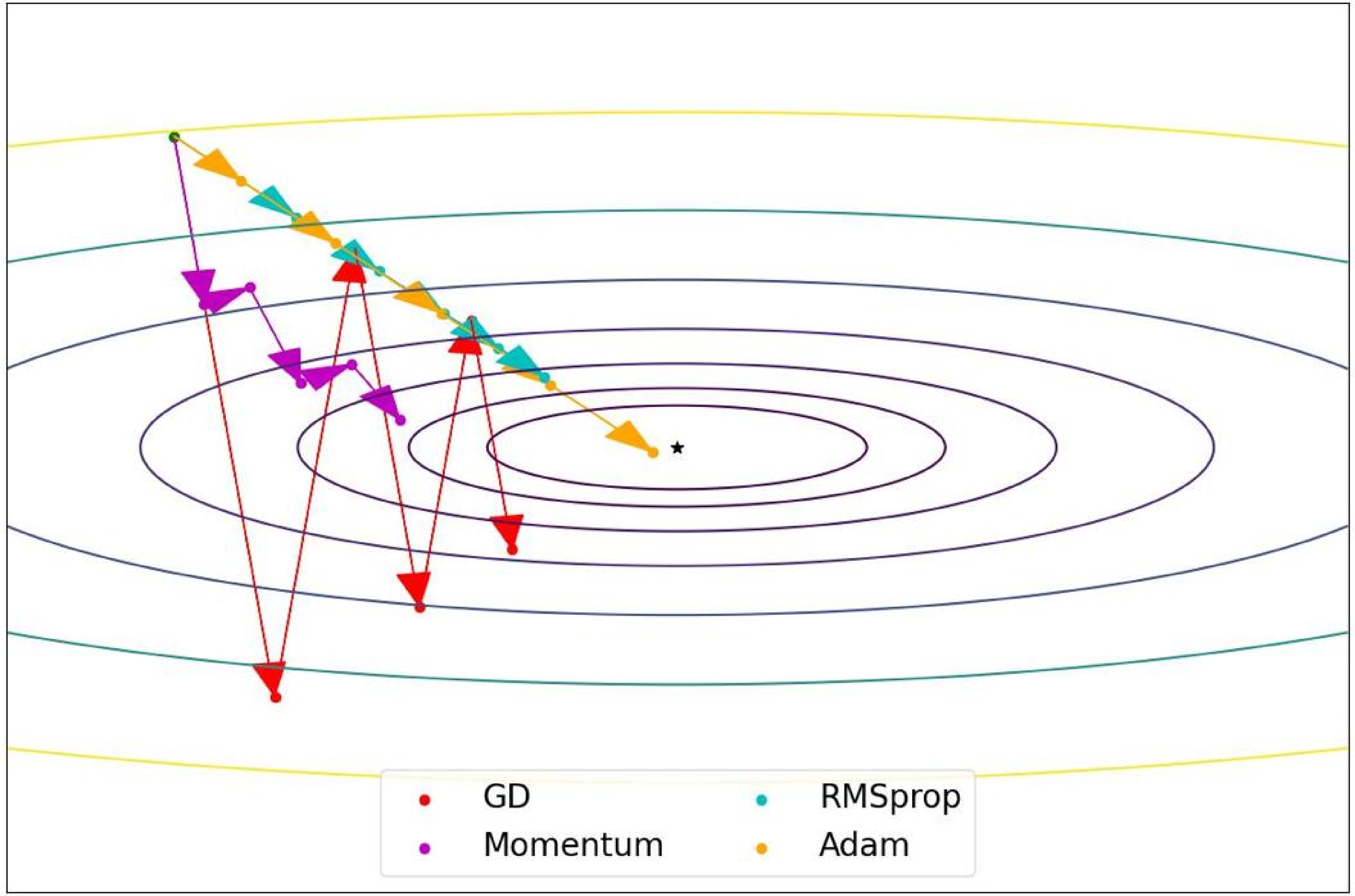
1. Computation details for $\hat{\boldsymbol{s}}_b^{[1](t+1)}$ and $\hat{\boldsymbol{v}}_b^{[1](t+1)}$

$$\boldsymbol{s}_b^{[1](t+1)} = \beta_2 \boldsymbol{s}_b^{[1](t)} + (1-\beta_2)\mathrm{d}\boldsymbol{b}^{[1](t)} \circ \mathrm{d}\boldsymbol{b}^{[1](t)} \qquad \hat{\boldsymbol{s}}_b^{[1](t+1)} = \frac{\boldsymbol{s}_b^{[1](t+1)}}{1-\beta_2^{t+1}}$$

$$\boldsymbol{v}_b^{[1](t+1)} = \beta_1 \boldsymbol{v}_b^{[1](t)} + (1-\beta_1)\mathrm{d}\boldsymbol{b}^{[1](t)} \qquad \hat{\boldsymbol{v}}_b^{[1](t+1)} = \frac{\boldsymbol{v}_b^{[1](t+1)}}{1-\beta_1^{t+1}}$$

- $\boldsymbol{v}_b^{[1](0)} = \boldsymbol{s}_b^{[1](0)} = 0$
- $\beta_1 = 0.9$ by default
- $\beta_2 = 0.99$ by default

Comparison



Comparison

1. Adam performs the best
2. Adam is commonly used in practice

Learning rate decay

1. Intuition
   - As iteration goes, the parameters should get closer to the theoretical values
   - It is inefficient to use the SAME learning rate for all iterations
     - ▷ [Inefficient] Small learning rate guarantees good performance, but it takes longer to get conver
     - ▷ [Unstable] A large learning rate leads to unstability when iteration index is large
   - Actually, we should decrease the learning rate in a reasonable manner

Learning rate decay

1. Denote
   - t : iteration index
   - $epoch$: epoch index
2. Several possible ways to decay the learning rate

$$\alpha_t = \frac{\alpha_0}{1 + \gamma \cdot epoch}$$
$$\alpha_t = \frac{\alpha_0}{\sqrt{epoch}}$$
$$\alpha_t = 0.95^{epoch} \cdot \alpha_0$$
$$\alpha_t = \frac{1}{\sqrt{t}} \cdot \alpha_0$$
$$\alpha_t = 0.95^t \cdot \alpha_0$$

   - $\alpha_0 = 5 \times 10^{-3}$ for example
   - $\gamma = 1$ (by default)