Learning rate schedulers and Optimisers

Perumadura De Silva

Content

- Gradient Descent Variants
  - Batch gradient descent
  - Stochastic gradient descent
  - Mini-batch gradient descent
- Learning rate scheduling methods
- Issues with vanilla stochastic gradient descent variants
- Gradient with momentum
- Adaptive gradient methods
- Gradient descent visualisation

Gradient Descent Variants

- Batch gradient descent:
  - Algorithm:
    - For e in epochs:
      - Compute the gradients given the dataset -> $\nabla J(\theta_e)$
      - Update parameters $\theta_{e+1} = \theta_e - \eta \nabla J(\theta_e)$ // η is the learning rate $0 < \eta < 1$
  - Uses the entire dataset to update the parameters
  - Issues:
    - Very slow since the entire dataset is used for the update
    - Intractable when dataset does not fit to memory
    - Can’t update online when new data comes

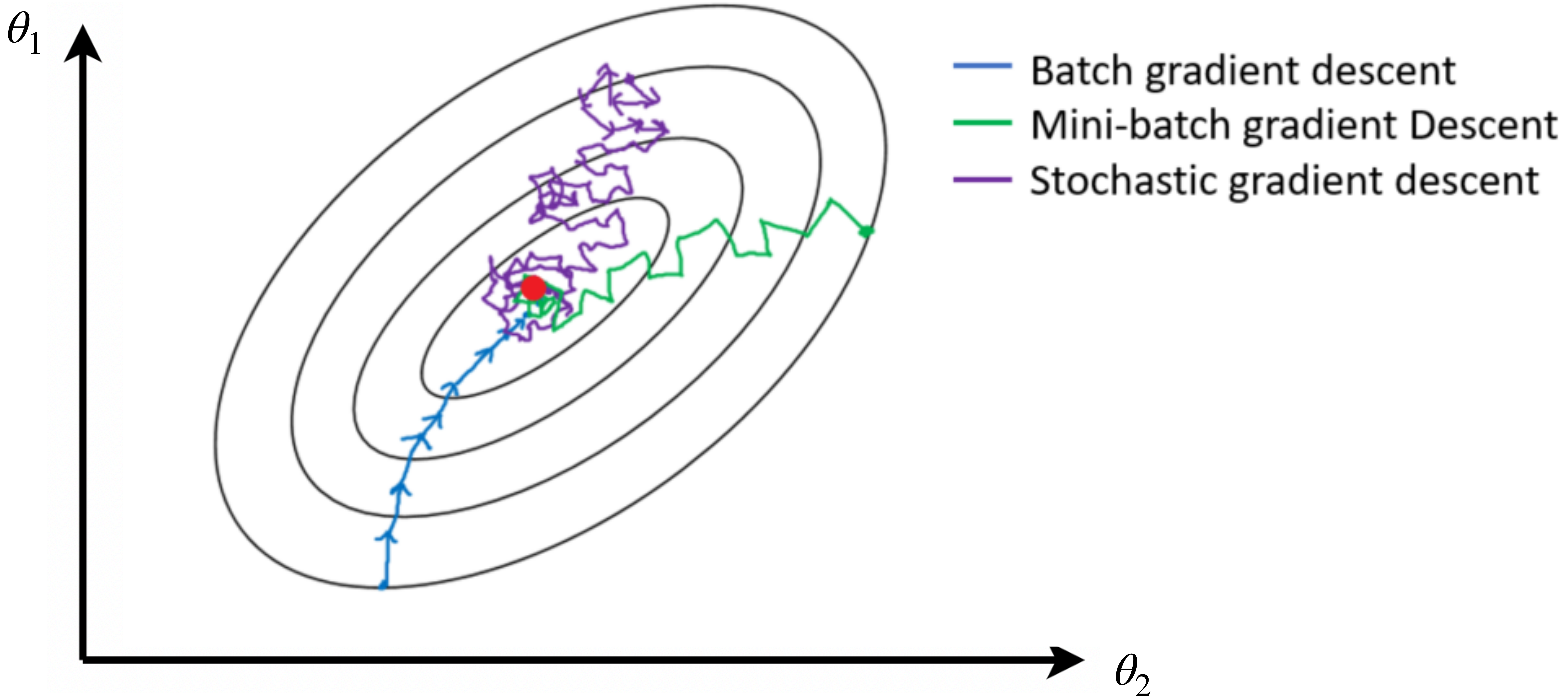
Gradient Descent Variants

- Stochastic gradient descent:
  - Instead of updating the parameters for the entire dataset, update for each datapoint
  - Algorithm:
    - For e in epochs:
      - For x_i, y_i in dataset:
        - Compute the gradients given the data point -> $\nabla J(\theta_i; x_i; y_i)$
        - Update parameters $\theta_{i+1} = \theta_i - \eta \nabla J(\theta_i; x_i, y_i)$
  - Parameters can be updated online when new data comes
  - Issues:
    - Since the parameter update is done for each datapoint, has a high variance in the parameter update
    - Jumping from one local minima to other with a constant learning rate η could lead to overshooting; learning rate schedulers address this problem

Gradient Descent Variants

- Mini-batch gradient descent:
  - Instead of updating on each sample, use a small batch of samples for the update; Reduces the variance in the update
  - Algorithm:
    - For e in epochs:
      - For $x_{i:i+n}, y_{i:i+n}$ in dataset: // n is the batch size
        - Compute the gradients given the dataset -> $\nabla J(\theta_{i:i+n}; x_{i:i+n}; y_{i:i+n})$
        - Update parameters $\theta_{(i:i+n)+1} = \theta_{i:i+n} - \eta \nabla J(\theta_{i:i+n}; x_{i:i+n}, y_{i:i+n})$
  - Online learning is possible when new data mini-batches arrives
  - Hardware parallelism can accelerate the computation in the mini-batch setup
  - Issues:
    - Similar to SGD, not guaranteed to converge; can stuck on a local minima; use LR-Scheduling to solve the issue (not completely)

Gradient Descent Variants



Ref.1: Convergence of different gradient descent methods

Learning rate scheduling methods

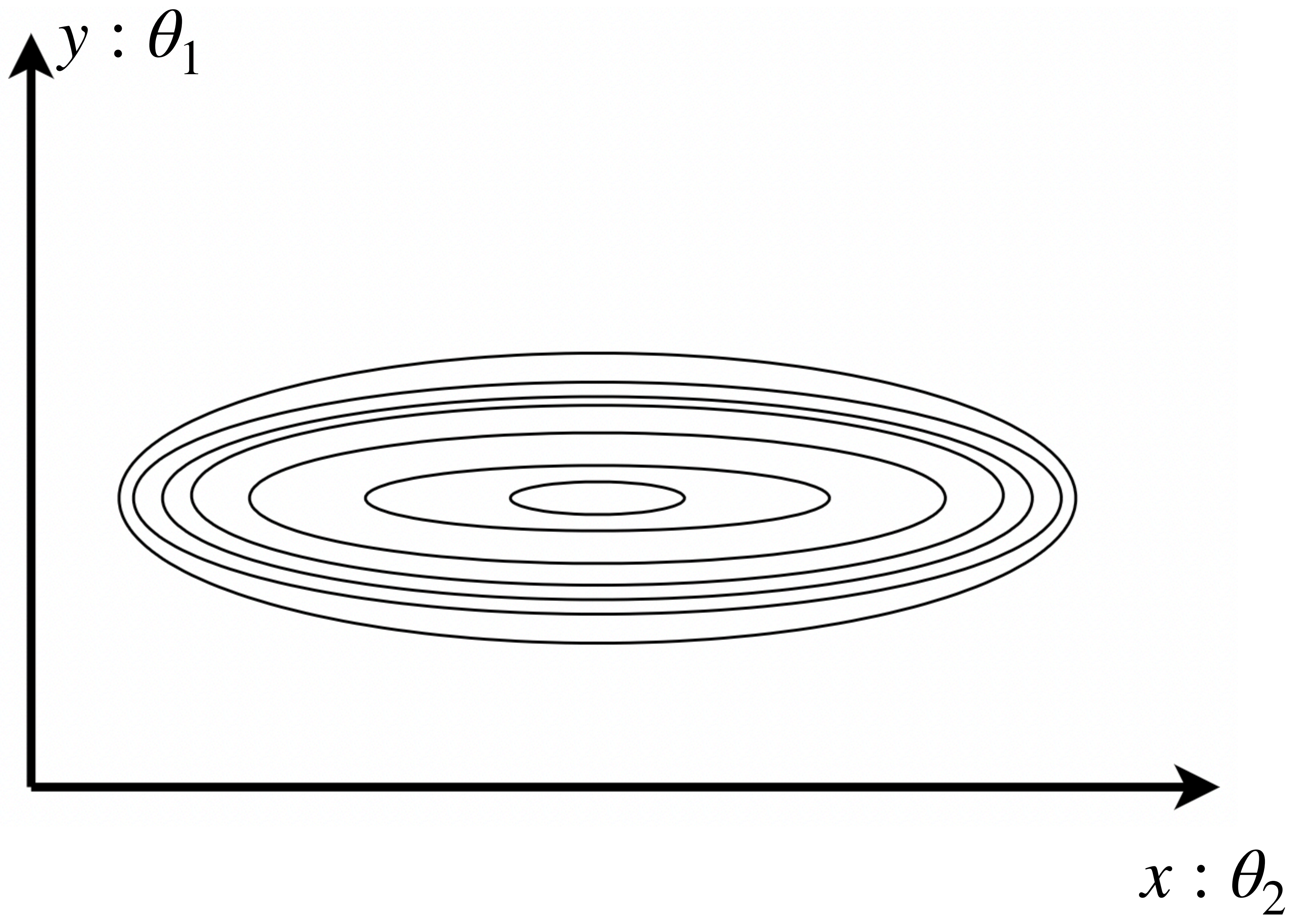
- As the loss decreases, the gradient of the loss surface reduces
  - When LR is constant, the gradients will bounce off the minima when the cost reaches zero
- So a schedular that reduces the learning rate at later steps can control the gradient step
- Different types of schedulers:
  - Polynomial decay: $\eta_t = \eta_0 * (t+1)^{\frac{1}{2}}$
  - Factor scheduler: $\eta_{t+1} = \eta_t * \alpha$ with a lower bound at $\eta_{t+1} = max(\eta_{min}, \eta_t * \alpha)$
  - Multifactor scheduler: $\eta_{t+1} = \eta_t * \alpha$ where $t \in S$; $S = \{10{,}20{,}30\}$; S set of epoch points
  - Cosine scheduler: $\eta_{t+1} = \eta_T + \frac{\eta_0 - \eta_t}{2}(1 + cos(\frac{\pi t}{T}))$ where $t \in [0{,}T]$
- LR Warmup: When models are very complex random initialisation of weights could lead to oscillations in performances; Have a warmup phase to stabilise this

Issues with vanilla stochastic gradient descent variants

- Very difficult to choose a learning rate
- To do learning rate scheduling, the scheduler structure must be properly determined:
  - Need an understanding about where the local minima occurs
  - When a momentum is applied to the gradient update, the suboptimal minima can be avoided
- When some data has low frequency features; applying the same LR could lead to slower low frequency feature extraction:
  - An adaptive learning rate applied to each neurone individually could solve this issue

Gradient momentum

- Momentum:
  - Idea: Use the moving average of the gradients instead of pure gradients for parameter update
  - y axis we want slower gradient updates and x axis faster
  - Compute the moving average for each parameter
    - $v_t = \gamma v_{t-1} + (1-\gamma)\nabla J(\theta_t)$
  - Update the parameters with averaged gradients
    - $\theta_{t+1} = \theta_t - \eta v_t$
  - Oscillation will be dampened when taking the moving average of (+) and (-) gradients
  - Gradients in the same direction will be encouraged: More momentum



Loss surface of a model with two parameters θ_1 and θ_2

Adaptive gradient methods

- Adagrad:
  - When there are features that occurs at different frequencies, it is suitable to apply an individual learning rate to each parameter
  - Adagrad achieves this by scaling the original learning rate by the accumulated gradient of that parameter
    - $\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{diag(G_t) + \epsilon I}} g_t$; ϵ to prevent the singularity
    - $g_t = \nabla J(\theta_t)$; parameter gradient at time step t
    - $G_t = \sum_{\tau=1}^{t} g_\tau g_\tau^T$; Accumulated squared gradient until time time step t
  - The adaptive gradient relaxes the choice of the initial learning rate
  - However positive accumulation of gradients through time could lead to a very small learning rate

Adaptive gradient methods

- RMSProp:
  - RMSProp and Adadelta are designed to address the gradient accumulation problem
  - RMSProp addresses this issue by computing the moving average of square gradients
  - moving average
    - $s_t = \rho s_{t-1} + (1 - \rho)[\nabla J(\theta_t)]^2$
  - LR is scaled by s_t for each parameter
    - $\theta_{t+1} = theta_t - \frac{\eta}{\sqrt{s_t + \epsilon I}} \nabla J(\theta_t)$

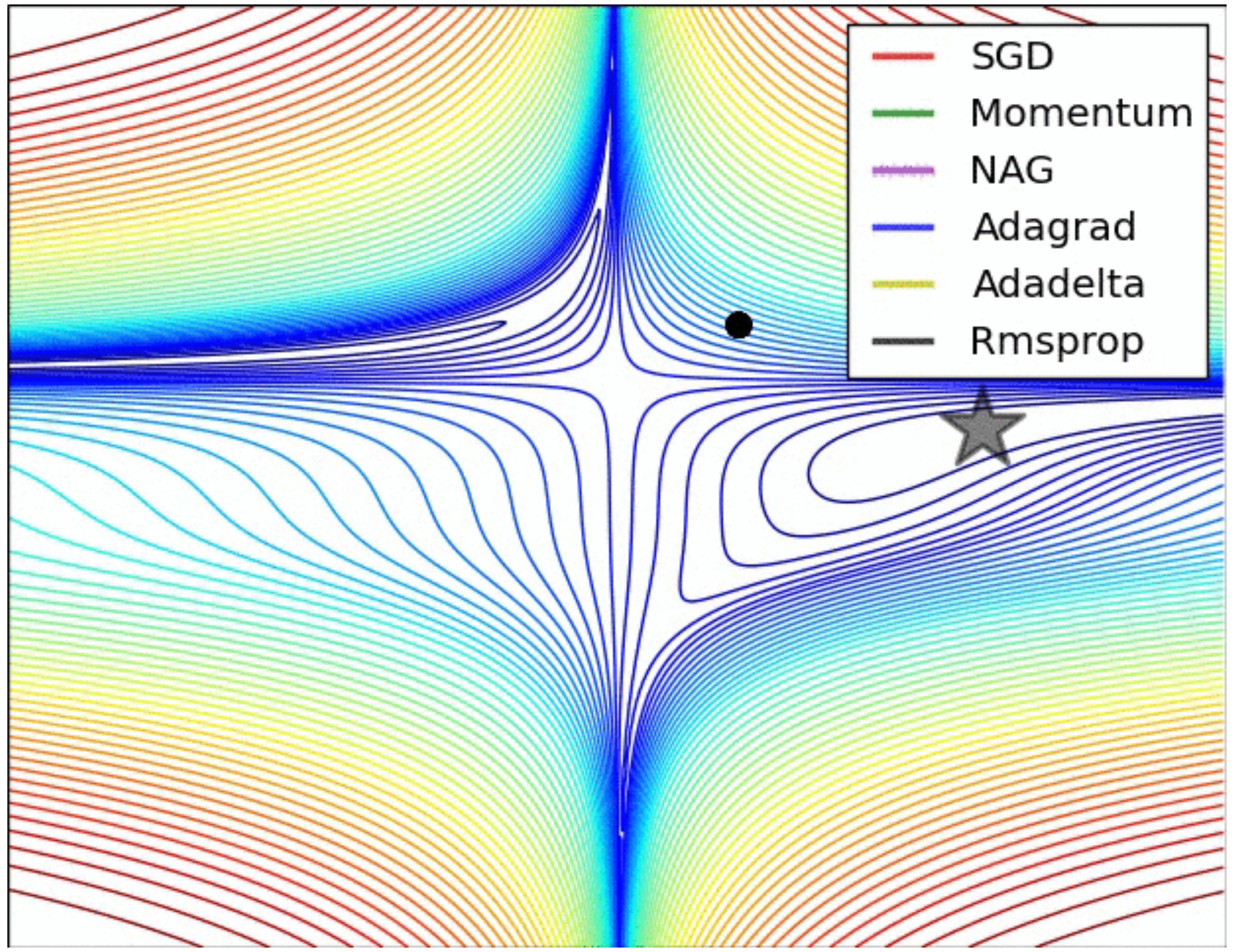
Adaptive gradient methods

- Adadelta:
  - With Adadelta, the choice of initial LR is not difficult and benefits from large initialisations
  - Instead of LR only, use the following expected moving average
    - $\mathbb{E}[\Delta\theta^2]_t = \rho\mathbb{E}[\Delta\theta^2]_{t-1} + (1-\rho)\Delta\theta_t^2$
  - Scale the learning rate by the expected squared gradients
    - $\mathbb{E}[g^2]_t = \rho\mathbb{E}[g^2]_{t-1} + (1-\rho)g_t^2$
    - $\Delta\theta = -\eta\frac{\sqrt{\mathbb{E}[\Delta\theta^2]_{t-1}}}{\sqrt{\mathbb{E}[g^2]_t + \epsilon I}}g_t$
    - $\theta_{t+1} = \theta_t + \Delta\theta_t$

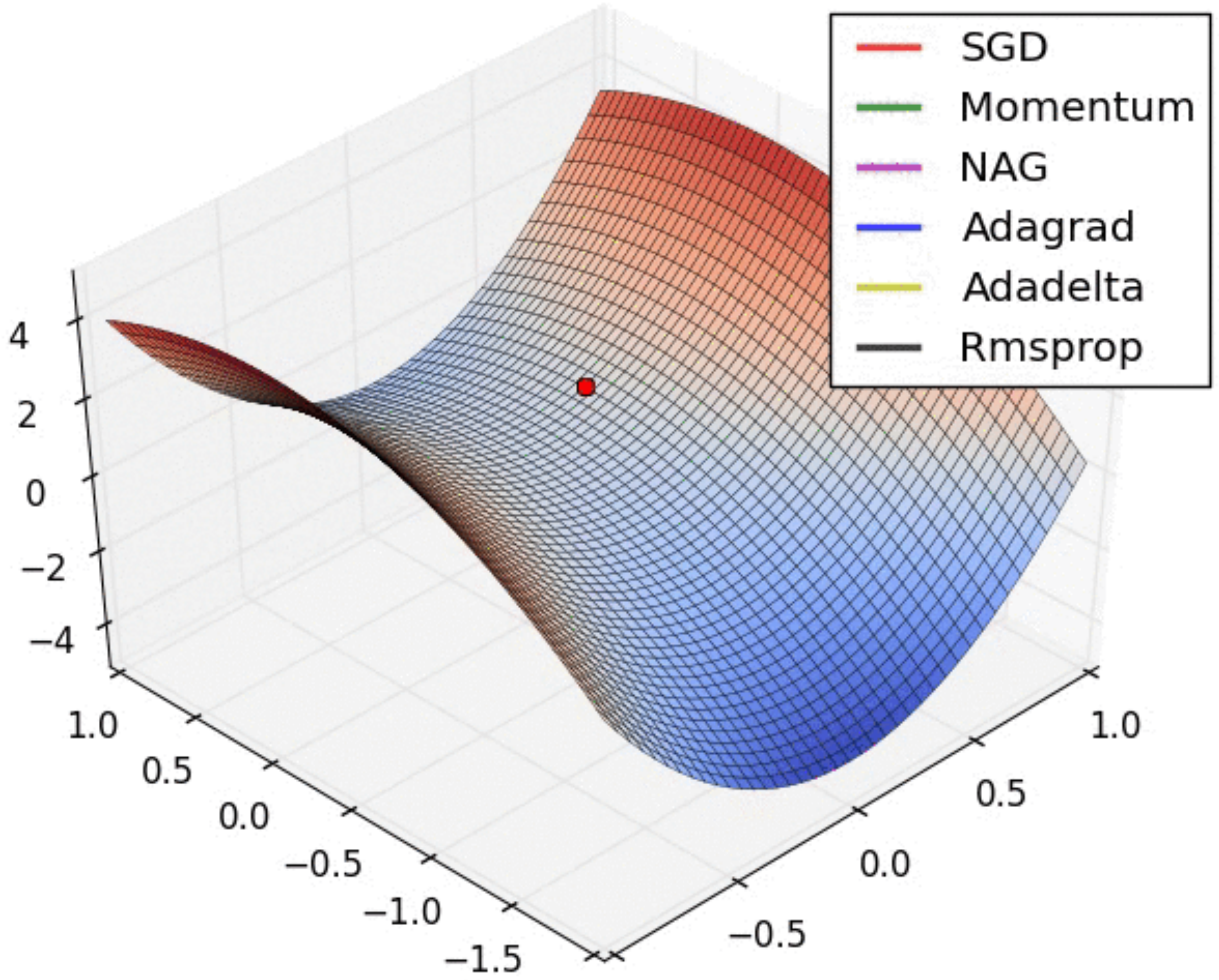
Adaptive gradient methods

- Adam:
  - Adam combines the benefits of both momentum and adaptive learning rate methods
  - Moving average momentum:
    - $v_t = \rho_1 v_{t-1} + (1 - \rho_1)[\nabla J(\theta_t)]$; Momentum damps oscillations
  - Moving average scaling factor:
    - $s_t = \rho_2 s_{t-1} + (1 - \rho_2)[\nabla J(\theta_t)]^2$; Adapt the learning rate for each parameters
  - When ρ_1, ρ_2 are large, v_t, s_t leans towards 0; specially at the early iterations
  - Use the following biased versions:
    - $\hat{v}_t = \dfrac{v_t}{1 - \rho_1^t}$ and $\hat{s}_t = \dfrac{s_t}{1 - \rho_2^t}$
  - Adaptive parameter update with a LR and momentum:
    - $\theta_{t+1} = \theta_t - \eta \dfrac{\hat{v}_t}{\sqrt{\hat{s}_t + \epsilon I}}$

Gradient descent visualisation



Ref.02: Gradient descent optimisation on a loss contour



Ref.02: Gradient descent on a saddle point

Reference

- Different gradient descent methods:
  - Ref.1: https://towardsdatascience.com/gradient-descent-algorithm-and-its-variants-10f652806a3
  - Ref.2: https://ruder.io/optimizing-gradient-descent/
  - https://blog.paperspace.com/optimization-in-deep-learning/
- On momentum method:
  - https://distill.pub/2017/momentum/
- Adaptive gradients:
  - https://medium.com/konvergen/an-introduction-to-adagrad-f130ae871827
  - https://towardsdatascience.com/10-gradient-descent-optimisation-algorithms-86989510b5e9