ECE 57000

Gradient Descent Algorithm

Chaoyue Liu

Fall 2024

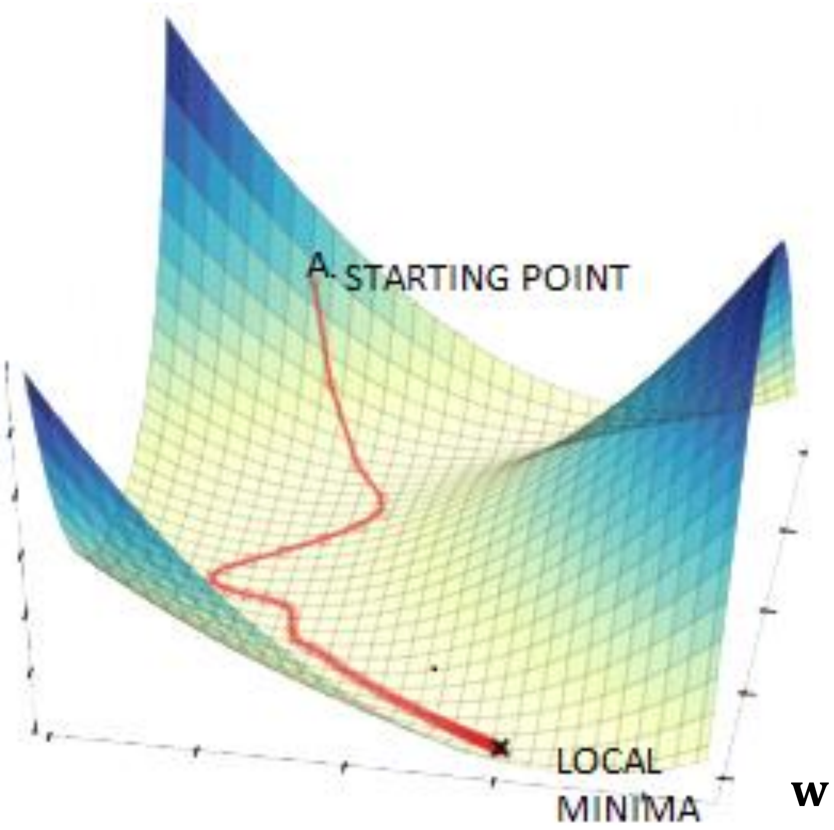
- Linear regression: MSE has a closed-form solution $\mathbf{w}^* = (X^T X)^{-1} X^T \boldsymbol{y}$
  - time complexity: $\sim O(d^3 + nd^2)$; space complexity: $\sim O(d^2)$
- Most models have **<u>no</u>** closed-form solutions

Must resort to numerical optimization

- The mostly used algorithm is **<u>gradient descent</u>** (or its variants, e.g., SGD)
  - intuitive
  - Relatively low computation cost
  - Parallel computable
- Other algorithms: Newton’s method, expectation–maximization (EM), …

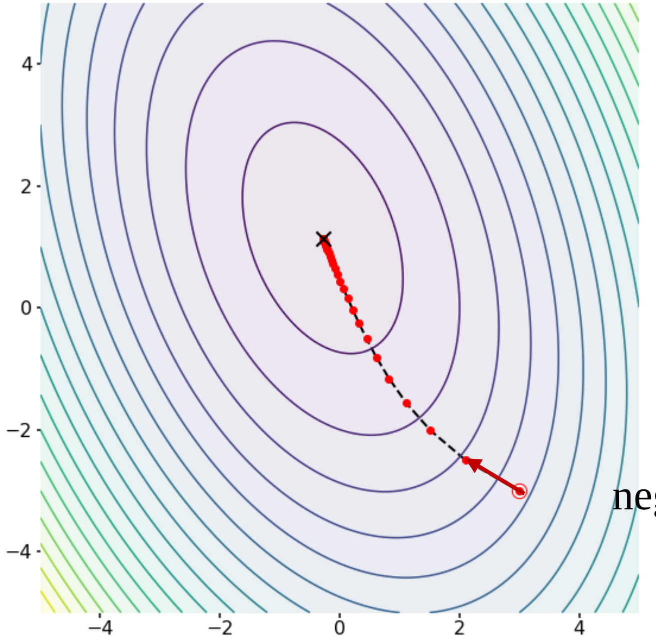
Gradient descent is like taking steps down the steepest descent into a valley

$\mathcal{L}(\mathbf{w})$



$\mathbf{w}$

w_1



neg. gradient
$-\nabla\mathcal{L}$

w_2

Contour plot

https://www.hackerearth.com/blog/developers/3-types-gradient-descent-algorithms-small-large-data-sets/

Gradient Descent

Goal: minimize the loss function (a.k.a. objective function) $\mathcal{L}(\mathbf{w})$

- i.e., find the $\mathbf{w}^*$ such that $\mathcal{L}(\mathbf{w}^*) \leq \mathcal{L}(\mathbf{w})$ for all $\mathbf{w}$

Step 1: Start with a guess of the weights $\mathbf{w}^0$ (can be random)

Step 2: evaluate the gradient of loss $\nabla\mathcal{L}(\mathbf{w}^t)$ at the current position $\mathbf{w}^t$ $(t \in \mathbb{N})$

Step 3: update parameter via <u>**negative gradient**</u> of loss function (η_t is step size or

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta_t \nabla\mathcal{L}(\mathbf{w}^t)$$

- η_t is called the <u>*learning rate*</u>

Step 4: repeat <u>Step 2 & 3</u> until convergence

Gradient computation

Gradient is basically the first-derivative:

$$\nabla \mathcal{L} = \frac{d\mathcal{L}}{d\mathbf{w}} = \left(\frac{\partial \mathcal{L}}{\partial w_1}, \frac{\partial \mathcal{L}}{\partial w_2}, \frac{\partial \mathcal{L}}{\partial w_3}, \cdots, \frac{\partial \mathcal{L}}{\partial w_p}\right)^T$$

For linear regression:

MSE loss is $\mathcal{L}(\mathbf{w}) = \frac{1}{2n}\sum_{i=1}^{n}\left(\mathbf{x}_i^T\mathbf{w} - y_i\right)^2 = \frac{1}{2n}\|X\mathbf{w} - \mathbf{y}\|_2^2$

$$\nabla \mathcal{L}(\mathbf{w}) = \frac{1}{n}\sum_{i=1}^{n}\underbrace{\left(\mathbf{x}_i^T\mathbf{w} - y_i\right)}_{\frac{d\ell}{df}} \cdot \underbrace{\mathbf{x}_i}_{\frac{df}{d\mathbf{w}}} = \frac{1}{n}X^{\mathrm{T}}(X\mathbf{w} - \mathbf{y})$$

$\boxed{\text{It is basically the } \textit{chain rule}\text{: } \frac{d\ell}{d\mathbf{w}} = \frac{d\ell}{df} \cdot \frac{df}{d\mathbf{w}}}$

Gradient computation

For <u>logistic regression</u>, the loss function is

$$\mathcal{L}(\mathbf{w}) = -\frac{1}{n}\sum_{i=1}^{n} y_i \log \sigma(\mathbf{w}^T\mathbf{x}_i) + (1 - y_i)\log\big(1 - \sigma(\mathbf{w}^T\mathbf{x}_i)\big)$$

A composition of these functions:

- $\ell_i(\mathbf{w}) = -y_i \log f(\mathbf{w}; \mathbf{x}_i) - (1 - y_i)\log(1 - f(\mathbf{w}; \mathbf{x}_i))$
- $f(\mathbf{w}; \mathbf{x}) = \sigma(\tilde{f}(\mathbf{w}; \mathbf{x}))$
- $\tilde{f}(\mathbf{w}; \mathbf{x}) = \mathbf{w}^T\mathbf{x}_i$

We can still use the *chain rule*:

$$\nabla\mathcal{L}(\mathbf{w}) \equiv \frac{d\mathcal{L}}{d\mathbf{w}} = \frac{1}{n}\sum\nolimits_{i=1}^{n} \frac{d\ell_i}{df} \cdot \frac{df}{d\tilde{f}} \cdot \frac{d\tilde{f}}{d\mathbf{w}}$$

$$= \frac{1}{n}\sum_{i=1}^{n} -\left(y_i \frac{1}{f(\mathbf{w}; \mathbf{x}_i) \cdot (1 - f(\mathbf{w}; \mathbf{x}_i))} - \frac{1}{1 - f(\mathbf{w}; \mathbf{x}_i)}\right) \cdot f(\mathbf{w}; \mathbf{x}_i) \cdot \big(1 - f(\mathbf{w}; \mathbf{x}_i)\big) \cdot \mathbf{x}_i$$

$$= \frac{1}{n}\sum_{i=1}^{n} (f(\mathbf{w}; \mathbf{x}_i) - y_i) \cdot \mathbf{x}_i$$

Gradient Descent

Goal: minimize the loss function (a.k.a. objective function) $\mathcal{L}(\mathbf{w})$

- i.e., find the $\mathbf{w}^*$ such that $\mathcal{L}(\mathbf{w}^*) \leq \mathcal{L}(\mathbf{w})$ for all $\mathbf{w}$

Step 1: Start with a guess of the weights $\mathbf{w}^0$ (can be random)

Step 2: evaluate the gradient of loss $\nabla\mathcal{L}(\mathbf{w}^t)$ at the current position $\mathbf{w}^t$ $(t \in \mathbb{N})$

Step 3: update parameter via **negative gradient** of loss function (η_t is step size or

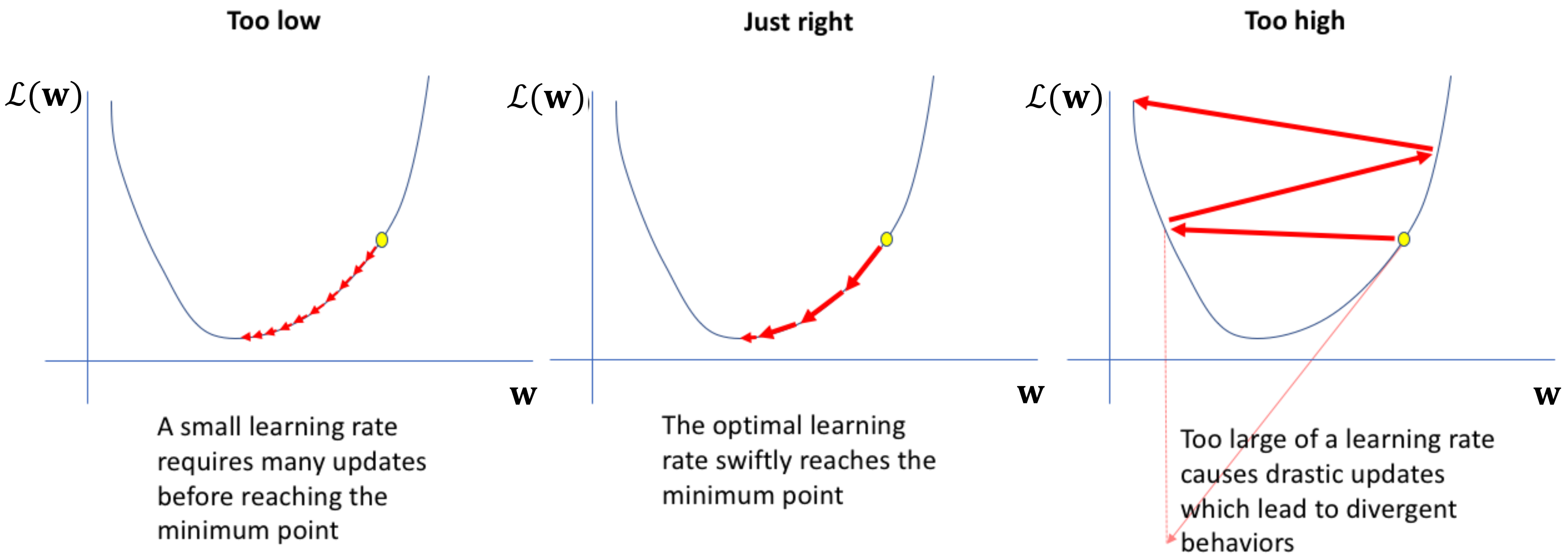
$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta_t \nabla\mathcal{L}(\mathbf{w}^t)$$

- η_t is called the *learning rate*

Step 4: repeat Step 2 & 3 until convergence

Learning rate η

- If learning rate is **too high**, the algorithm could diverge.
  - Diverge means to get farther away from the solution.
- If learning rate **too low**, the algorithm could take a very long time to converge.



https://www.jeremyjordan.me/nn-learning-rate/

Stochastic Gradient Descent (SGD)

GD can be computationally costly: $\sim O(n)$ per iteration

$$\mathcal{L}(\mathbf{w};\mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n} \ell(\mathbf{w};\mathbf{x}_i,y_i) \longrightarrow \nabla\mathcal{L}(\mathbf{w};\mathcal{D}) = \frac{1}{n}\sum_{i=1}^{n} \nabla\ell(\mathbf{w};\mathbf{x}_i,y_i)$$

- when training set size n is large, huge cost for each iteration

SGD: Use <u>one</u> training sample to estimate the full gradient

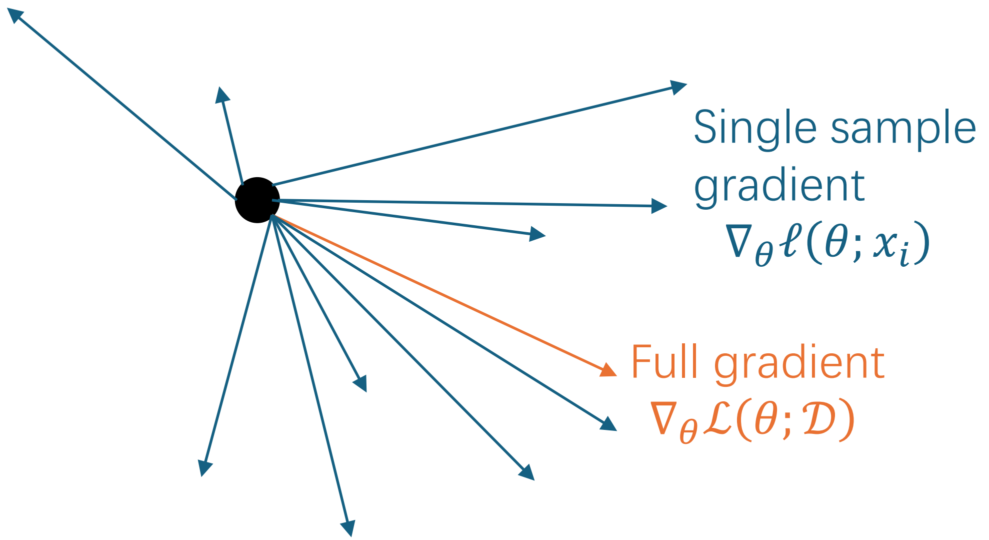
$$\nabla\ell(\mathbf{w};\mathbf{x}_i,y_i) \approx \nabla\mathcal{L}(\mathbf{w};\mathcal{D})$$

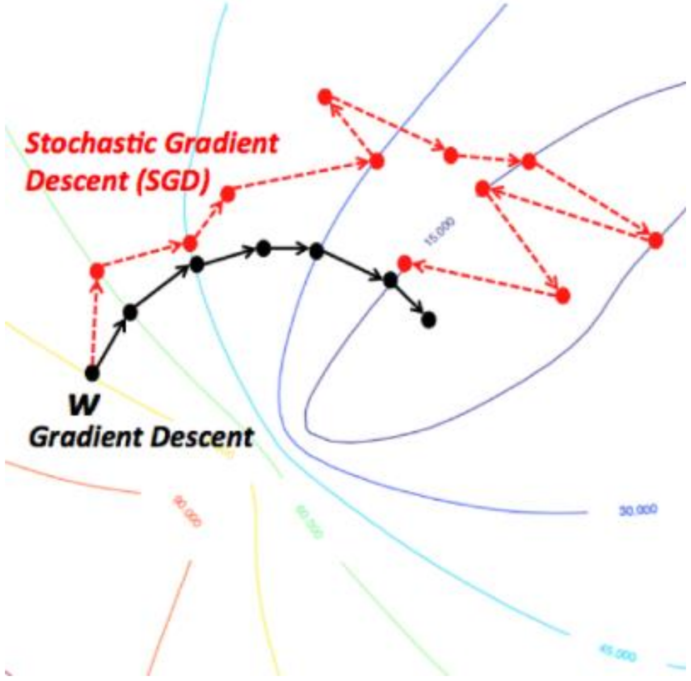
- update rule: $\mathbf{w}^{t+1} = \mathbf{w}^{t} - \eta_t \nabla\ell(\mathbf{w}^{t};\mathbf{x}_i,y_i)$
- the data $(\mathbf{x}_i,y_i)$ is <u>randomly</u> sampled in each update
- <u>no bias introduced</u>: $\mathbb{E}_i[\nabla\ell(\mathbf{w};\mathbf{x}_i,y_i)] = \nabla\mathcal{L}(\mathbf{w};\mathcal{D})$

Stochastic Gradient Descent (SGD)

Although no bias, stochastic gradient introduces noise

$$\nabla \ell(\mathbf{w}; \mathbf{x}_i, y_i) = \nabla \mathcal{L}(\mathbf{w}; \mathcal{D}) + \xi_i$$





Full gradient is average over single sample gradients. This is why it is “stochastic”.

mini-batch SGD

One sample may be too noisy, why not use several samples to estimate?

Randomly sample b training data points: $\{(\mathbf{x}_{i_1}, y_{i_1}), (\mathbf{x}_{i_2}, y_{i_2}), \cdots, (\mathbf{x}_{i_b}, y_{i_b})\}$

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta_t \frac{1}{b} \sum_{k=1}^{b} \nabla \ell(\mathbf{w}; \mathbf{x}_{i_k}, y_{i_k})$$

- Intermediate level computation cost per iteration: $\sim O(b)$
- Less noisy than SGD

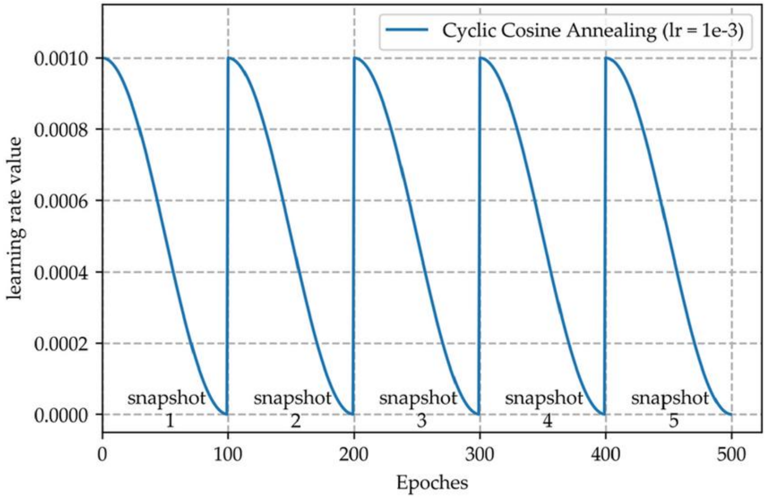
One pass (a.k.a. epoch) through dataset

- GD: 1 update
- SGD: n updates
- Mini-batch SGD: $\frac{n}{b}$ updates

Learning rate η_t

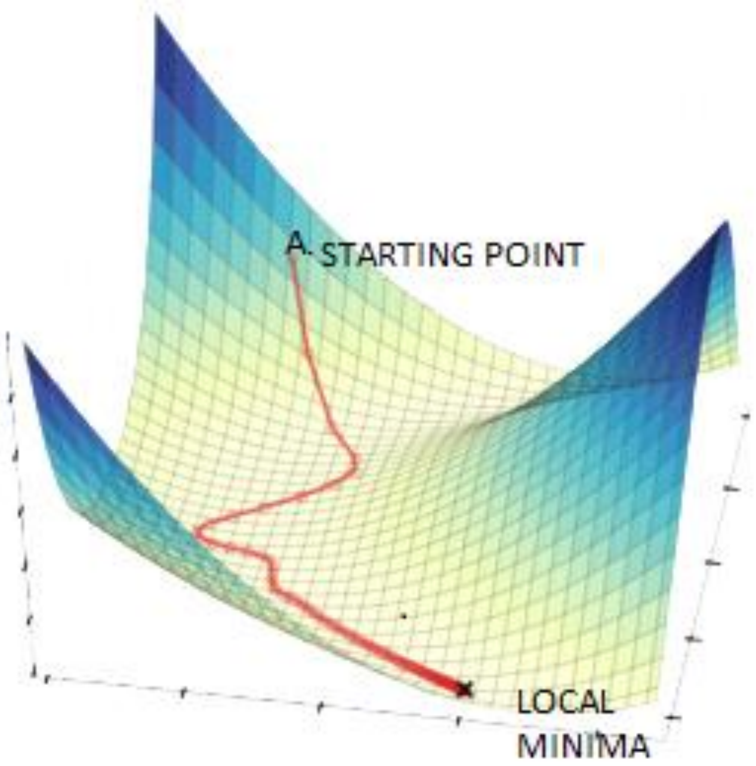
The learning rate does not have to be a constant

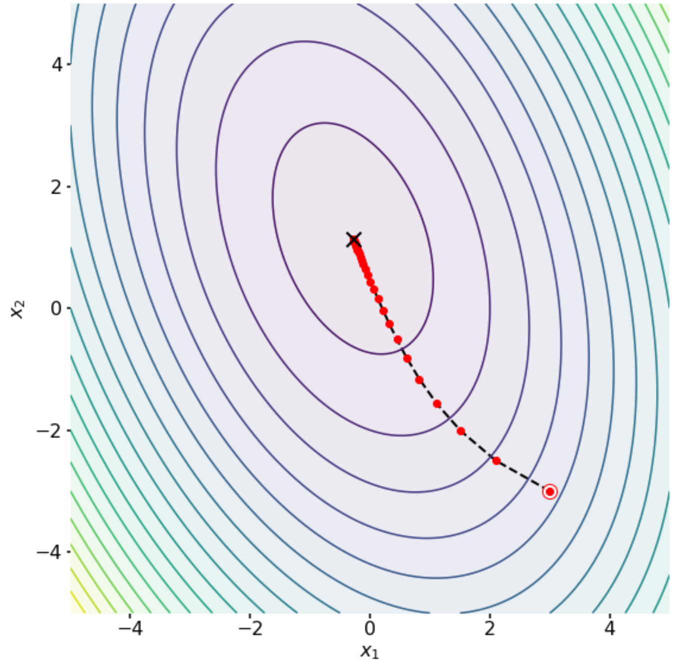
- decreasing step sizes, for example $\eta_t = \frac{1}{t}$
  - Sum of η_t has to cover an infinite distance, $\lim_{T \to \infty} \sum_{t=1}^{T} \eta_t = \infty$
  - Helps to reduce stabilize SGD near convergence
- Cosine scheduler is another possibility, which works well in some cases



Momentum method

- Imagine putting a small metal ball at starting point A, and set it free
- The ball accelerates, due to accumulation of <u>momentum</u>





Contour plot

Momentum method

Heavy ball momentum :

momentum: $\mathbf{m}^t = -\nabla\mathcal{L}(\mathbf{w}^t) + \beta \cdot \mathbf{m}^{t-1}$

parameters: $\mathbf{w}^{t+1} = \mathbf{w}^t + \eta \cdot \mathbf{m}^t$

learning rate: η
momentum parameter: β
$(0 < \beta < 1)$

Nesterov momentum:

look ahead: $\tilde{\mathbf{w}}^t = \mathbf{w}^t + \eta\beta\mathbf{m}^{t-1}$

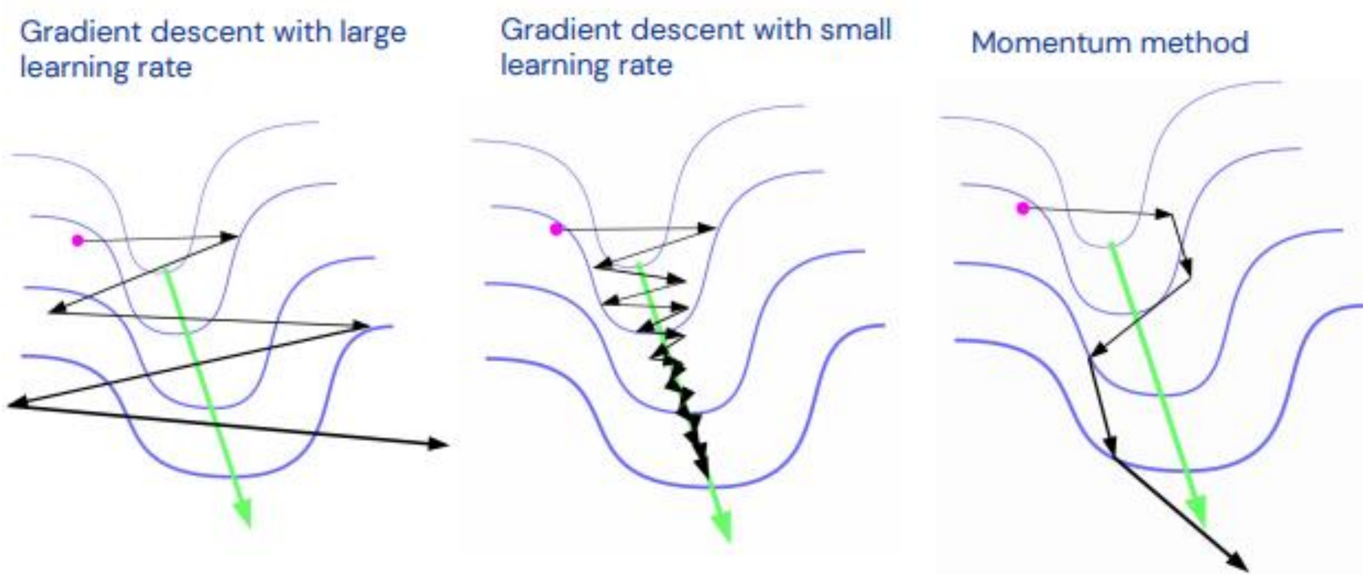
momentum: $\mathbf{m}^t = -\nabla\mathcal{L}(\tilde{\mathbf{w}}^t) + \beta \cdot \mathbf{m}^{t-1}$

parameters: $\mathbf{w}^{t+1} = \mathbf{w}^t + \eta \cdot \mathbf{m}^t$

learning rate: η
momentum parameter: β
$(0 < \beta < 1)$

- These momentum methods are proven to accelerate training for convex problems
- Can be used together with SGD/mini-batch SGD (not guaranteed to accelerate)

Momentum method



Implementation:

- (heavy ball) momentum: `optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9)`
- Nesterov momentum: `optimizer = optim.SGD(model.parameters(), lr=0.01, momentum=0.9, Nesterov=True)`